前言:工欲善其事必先利其器,想要彻底使用一门技术，先会使用它是一种基本的学习过程,切勿好高骛远，眼高手低。本文先介绍springMvc的基本是使用过程,后面有专门讲其源码的分析文章。后面再说。

注:

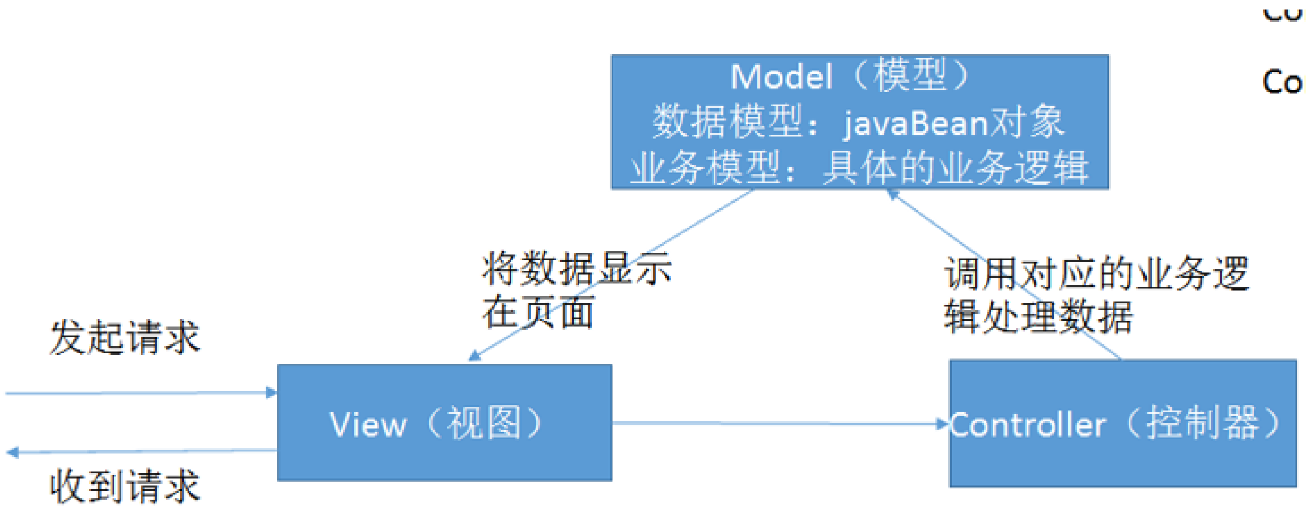
1.本文统一使用IDEA作为开发工具以及调试工具

2.springMvc的版本统一使用

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.3.17.RELEASE</version>
</dependency>
```

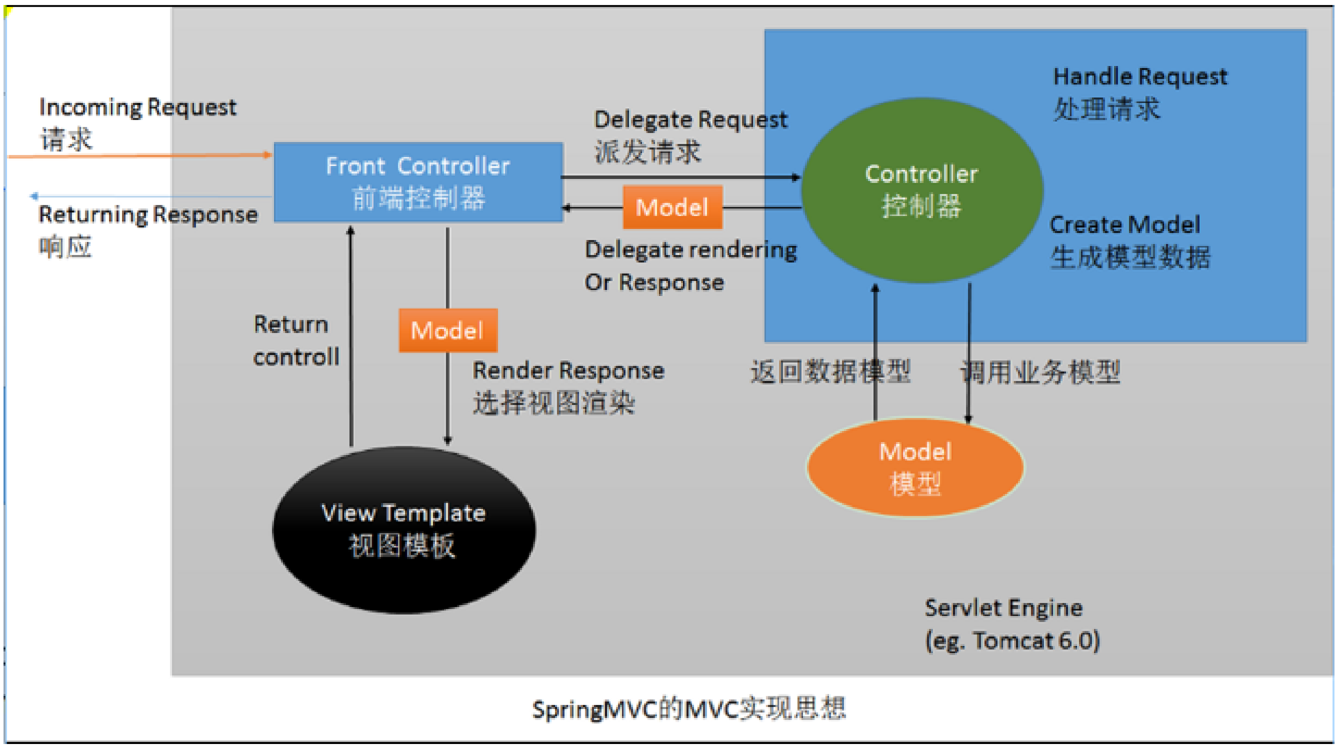
1.mvc简介

1.标准的mvc模式示意图:



这是一个标准的mvc模式图。

2.springMvc自己理解的mvc模式图



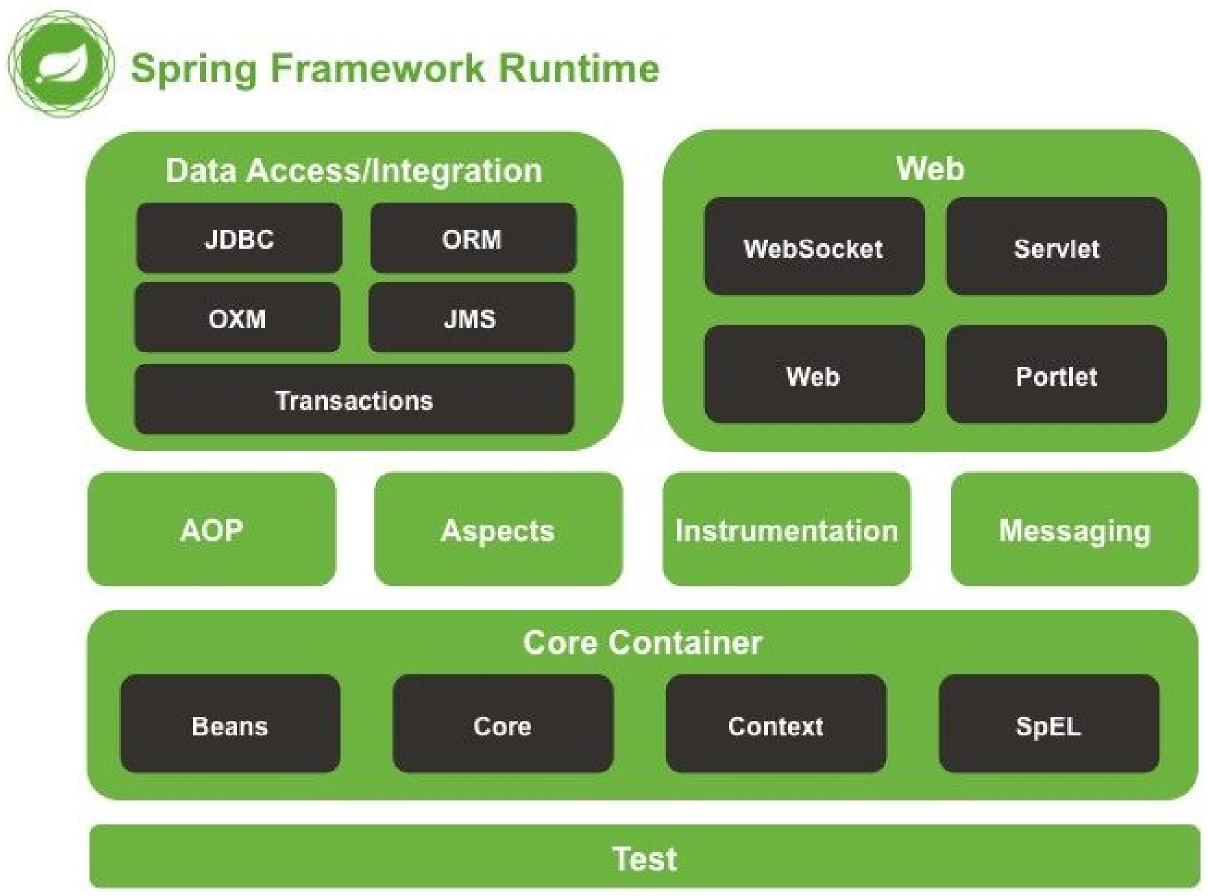
springMvc更加强调模块性，此点后面讲源码时会更加凸显。这是springMvc自己理解的mvc模式的设计图。它独特增加了一个前端控制器，此对应于springMvc中的DispatcherServlet。这个类是不是很熟悉？没错就是我们通常在web.xml中做的那个配置。举个栗子，前端控制器相当于一个医院的导诊台，它具有指导的作用。

2.小栗子

先举个小栗子做下说明springMvc项目的基本使用。

1.导包

```
核心容器的包：
spring-beans-4.0.0.RELEASE.jar
spring-context-4.0.0.RELEASE.jar
spring-core-4.0.0.RELEASE.jar
spring-expression-4.0.0.RELEASE.jar
web的包：
spring-web-4.0.0.RELEASE.jar 这个包是与原生的web整合
spring-webmvc-4.0.0.RELEASE.jar  这个包是springMvc包
aop包，主要是注解需要使用：
spring-aop-4.0.0.RELEASE.jar
日志包：
commons-logging-1.1.3.jar
```



注:spring官网上提供的一张整个spring framework的整体图,上层依赖下层，越到上层依赖越大。这张图理解的越深越好。

2.基本配置

web.xml的配置

```
<web-app>
  <servlet>
    <servlet-name >spring-mvc</servlet-name>
    <!-- servlet类 -->
    <servlet-class >org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 初始化参数 -->
    <init-param >
      <!--如果没有配置文件,那么默认会去web应用的/WEB-INF下找一个名叫 '前端控制器名-servlet.xml' 的默认配置文件
-->
      <!--如果此例中没有配置那么默认名称应该是/WEB-INF/spring-mvc-servlet.xml的文件 -->
      <!--如果不想指定配置文件那么就在/WEB-INF/创建一个spring-mvc-servlet.xml的文件-->
      <param-name >contextConfigLocation</param-name>
      <param-value >classpath:/spring-mvc.xml</param-value>
    </init-param>
    <!-- 启动时加载的优先级,值越小优先级越高 -->
    <load-on-startup >1</load-on-startup>
  </servlet>
  <servlet-mapping >
    <servlet-name >spring-mvc</servlet-name>
    <!-- /与/*的差别如下-->
    <!-- / 拦截所有请求，除了jsp  /* 拦截一切请求包括jsp -->
    <!-- 这里写/ 其实服务器是无法处理静态请求的，这里要特别注意 那么该如何处理让其可以处理静态资源呢？见下面的解决方案 -->
    <!--
        /：拦截所有请求，不拦截jsp页面，*.jsp请求
        /*：拦截所有请求，拦截jsp页面，*.jsp请求
        处理*.jsp是tomcat做的事；所有项目的小web.xml都是继承于大web.xml
        DefaultServlet是Tomcat中处理静态资源的？
            除过jsp，和servlet外剩下的都是静态资源；
            index.html xxx.js：静态资源，tomcat就会在服务器下找到这个资源并返回；
            我们前端控制器的/禁用了tomcat服务器中的DefaultServlet

        1）服务器的大web.xml中有一个DefaultServlet是url-pattern=/
        2）我们的配置中前端控制器 url-pattern=/
                静态资源会来到DispatcherServlet（前端控制器）看那个方法的RequestMapping是这个index.html
        3）为什么jsp又能访问；因为我们没有覆盖服务器中的JspServlet的配置
        4）  /*  直接就是拦截所有请求；我们写/；也是为了迎合后来Rest风格的URL地址
    -->
    <url-pattern >/</url-pattern>
  </servlet-mapping>
</web-app>
上面已经说了为何/ 拦截所有请求，除了jsp  而/* 拦截一切请求包括jsp的原因
此处展示一下大web.xml的部分配置
在apache-tomcat-7.0.82\conf配置文件路径下的web.xml的总web.xml里面，所有应用的web.xml全都继承这个总的xml文件
在这个文件里有这么一段配置：
<servlet>
    <!-- The default servlet for all web applications, that serves static resources -->
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <init-param>
        <param-name>listings</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>fork</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>xpoweredBy</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>
<servlet-mapping>
```

```
    <servlet-name>jsp</servlet-name>
    <url-pattern>*.jsp</url-pattern>
    <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

让上面配置让springMvc拦截/ 打破了tomcat处理静态资源的能力,所以这里提供处理静态资源的解决方法，在spring-mvc.xml中:

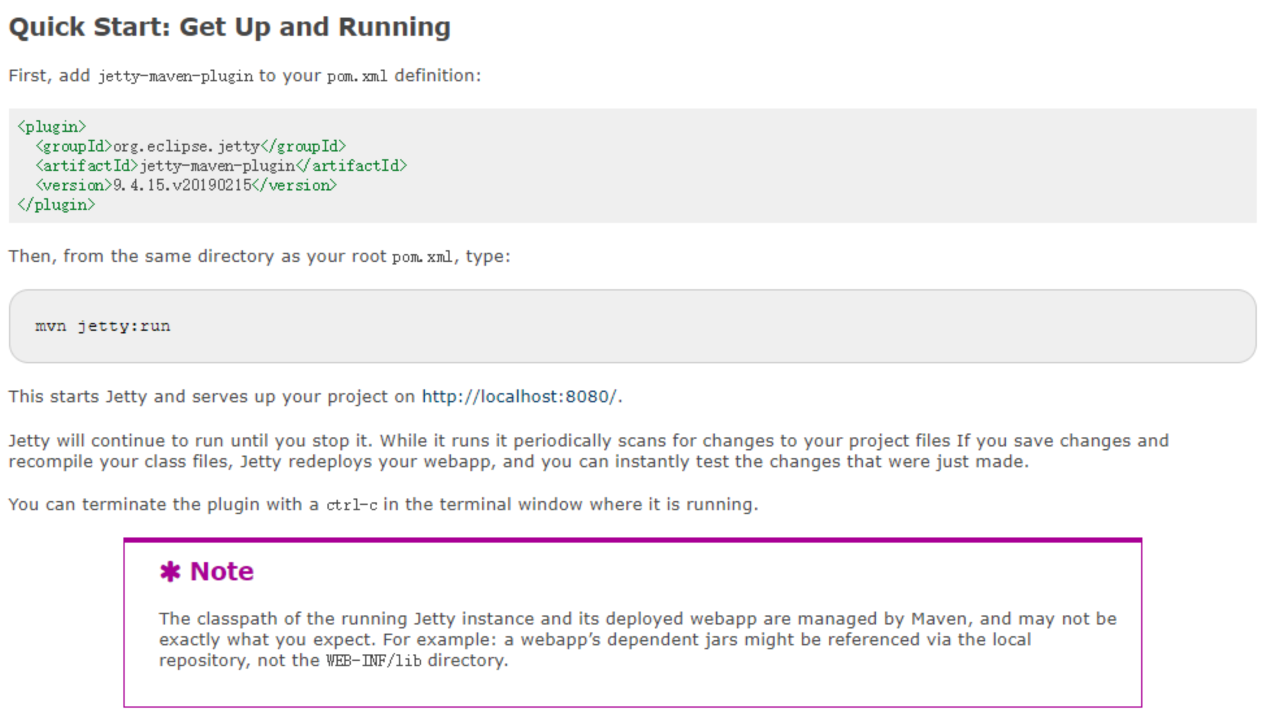
```
<!-- 默认前端控制器是拦截所有资源（除过jsp），js文件就404了；要js文件的请求是交给tomcat处理的
    http://localhost:8080/xxx/scripts/jquery-1.9.1.min.js
    -->
<!-- 告诉SpringMVC，自己映射的请求就自己处理，不能处理的请求直接交给tomcat -->
<!-- 静态资源能访问，动态映射的请求就不行 所以还得加上下面的配置 -->
<mvc:default-servlet-handler/>
<!-- springmvc可以保证动态请求和静态请求都能访问 -->
<mvc:annotation-driven></mvc:annotation-driven>
```

因为在IDEA中实在不想配置Tomcat,实际上是可以的，我试过了确实配置一个本地的Tomcat。现在我说下我是咋不配置Tomcat而使用web容器的。我用的是jetty。以plugin的形式运行。

pom.xml配置jetty

```
<build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.eclipse.jetty</groupId>
          <artifactId>jetty-maven-plugin</artifactId>
          <version>9.4.15.v20190215</version>
        </plugin>
      </plugins>
    </pluginManagement>
</build>
```

为什么这么配置呢？见jetty官网https://www.eclipse.org/jetty/documentation/current/jetty-maven-plugin.html#get-up-and-running



Quick Start: Get Up and Running

First, add `jetty-maven-plugin` to your `pom.xml` definition:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.4.15.v20190215</version>
</plugin>
```

Then, from the same directory as your root `pom.xml`, type:

```
mvn jetty:run
```

This starts Jetty and serves up your project on http://localhost:8080/.

Jetty will continue to run until you stop it. While it runs it periodically scans for changes to your project files If you save changes and recompile your class files, Jetty redeploys your webapp, and you can instantly test the changes that were just made.

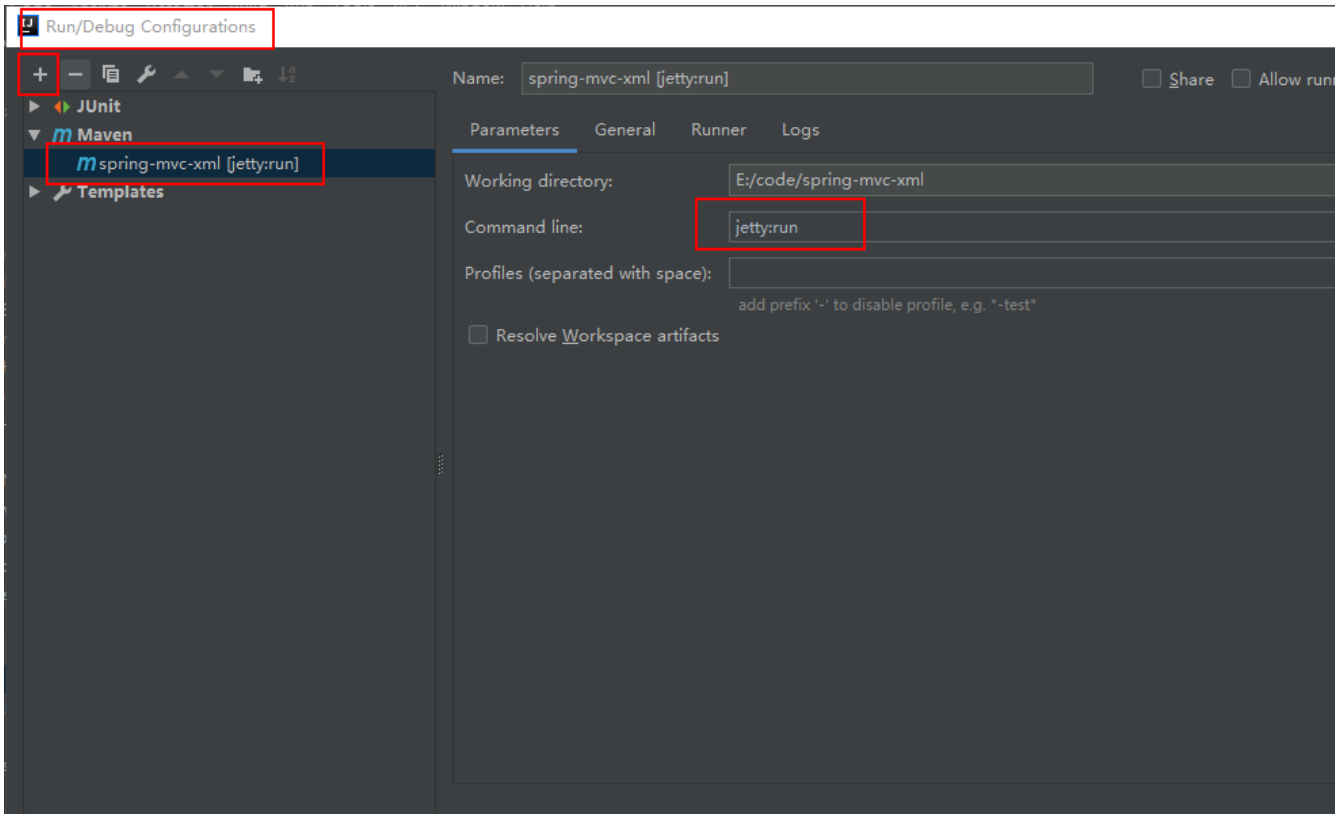
You can terminate the plugin with a `ctrl-c` in the terminal window where it is running.

✱ Note

The classpath of the running Jetty instance and its deployed webapp are managed by Maven, and may not be exactly what you expect. For example: a webapp's dependent jars might be referenced via the local repository, not the `WEB-INF/lib` directory.

然后到与pom.xml文件同级的目录下使用命令行运行mvn jetty:run 就可运行整个web项目了

当你足够熟悉了mvn后其实也是可以直接配置在IDEA上的如下图:点击+号，调出Maven 即可进行配置



Name: spring-mvc-xml [jetty:run]

spring-mvc-xml [jetty:run]

Working directory: E:/code/spring-mvc-xml

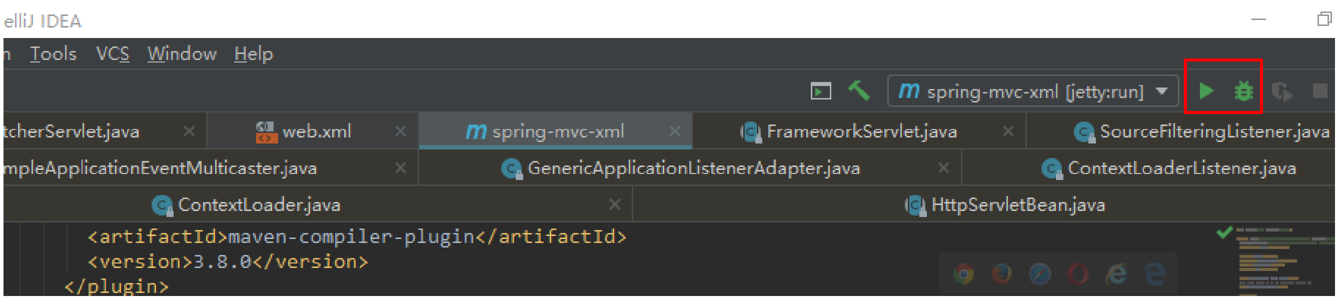
Command line: jetty:run

Profiles (separated with space):

add prefix '-' to disable profile, e.g. "-test"

☐ Resolve Workspace artifacts

此时点击运行或者调试按钮可以进行运行或者调试。


elliJ IDEA
n Tools VCS Window Help
spring-mvc-xml [jetty:run]
tcherServlet.java
web.xml
spring-mvc-xml
FrameworkServlet.java
SourceFilteringListener.java
mpleApplicationEventMulticaster.java
GenericApplicationListenerAdapter.java
ContextLoaderListener.java
ContextLoader.java
HttpServletBean.java

```
<artifactId>maven-compiler-plugin</artifactId>
<version>3.8.0</version>
</plugin>
```

注:可以改默认端口号jetty的默认端口号是8080，

方法: mvn -Djetty.http.port=9999 jetty:run 这个命令就可以将8080端口改为9999端口

```
jetty官网上摘抄的一段话:
httpConnector
Optional. If not specified, Jetty will create a ServerConnector instance listening on port 8080.
You can change this default port number by using the system property jetty.http.port on the
command line, for example, mvn -Djetty.http.port=9999 jetty:run. Alternatively, you can use this
configuration element to set up the information for the ServerConnector. The following are the
valid configuration sub-elements:

port
The port number for the connector to listen on. By default it is 8080.
host
The particular interface for the connector to listen on. By default, all interfaces.
name
The name of the connector, which is useful for configuring contexts to respond only on particular
connectors.
idleTimeout
Maximum idle time for a connection.

You could instead configure the connectors in a standard jetty xml config file and put its
location into the jettyXml parameter. Note that since Jetty 9.0 it is no longer possible to
configure a https connector directly in the pom.xml: you need to use jetty xml config files to do
it.
```

配置spring-mvc.xml

```
<context:component-scan base-package="com.mgw"></context:component-scan>
<!-- 视图解析器，可以简化方法的返回值，返回值就是作为目标页面地址，
    只不过视图解析器可以帮我们拼串
 -->
 <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/pages/"></property>
    <property name="suffix" value=".jsp"></property>
 </bean>
```

3.处理总逻辑

创建一个controller的包

创建一个处理器类

```
@Controller
public class HelloCroller {

    @RequestMapping("/hello")
    public String hello() {
        return "index";
    }
}
```

4.页面请求即可

```
分析这个小栗子的运行流程:
1) 、客户端点击链接会发送 http://localhost:8080/springmvc-xml/hello 请求
2) 、来到tomcat服务器;
3) 、SpringMVC的前端控制器收到所有请求;
4) 、来看请求地址和@RequestMapping标注的哪个匹配，来找到到底使用那个类的哪个方法来处理
5) 、前端控制器找到了目标处理器类和目标方法，直接利用反射执行目标方法;
6) 、方法执行完成以后会有一个返回值; SpringMVC认为这个返回值就是要去的页面地址
7) 、拿到方法返回值以后; 用视图解析器进行拼串得到完整的页面地址;
8) 、拿到页面地址，前端控制器帮我们转发到页面;
```

不可以两个方法处理一个请求，只能一个方法处理一个请求

5、@RequestMapping；就是告诉SpringMVC；这个方法用来处理什么请求； 这个/是可以省略，即使省略了，也是默认从当前项目下开始； 习惯加上比较好 /hello /hello

RequestMapping小细节

```
/**
 * RequestMapping的其他属性
 * method: 限定请求方式、
 *      HTTP协议中的所有请求方式:
 *          【GET】, HEAD, 【POST】, PUT, PATCH, DELETE, OPTIONS, TRACE
 *      GET、POST
 *      method=RequestMethod.POST: 只接受这种类型的请求，默认是什么都可以;
 *          不是规定的方式报错: 4xx:都是客户端错误
 *              405 - Request method 'GET' not supported
 * params: 规定请求参数
 * params 和 headers支持简单的表达式:
 *      param1: 表示请求必须包含名为 param1 的请求参数
 *          eg: params={"username"}:
 *              发送请求的时候必须带上一个名为username的参数; 没带就会404
 *
 *      !param1: 表示请求不能包含名为 param1 的请求参数
 *          eg:params={"!username"}
 *              发送请求的时候必须不携带上一个名为username的参数; 带了就会404
 *      param1 != value1: 表示请求包含名为 param1 的请求参数，但其值不能为 value1
 *          eg: params={"username!=123"}
 *              发送请求的时候;携带的username值必须不是123(不带username或者username不是123)
 *
 *      {"param1=value1", "param2"}: 请求必须包含名为 param1 和param2 的两个请求参数，且 param1 参数的值必须为 value1
 *          eg:params={"username!=123","pwd","!age"}
 *              请求参数必须满足以上规则;
 *              请求的username不能是123，必须有pwd的值，不能有age
```

```java
 * headers：规定请求头；也和params一样能写简单的表达式
 *
 *
 *
 * consumes：只接受内容类型是哪种的请求，规定请求头中的Content-Type
 * produces：告诉浏览器返回的内容类型是什么，给响应头中加上Content-Type:text/html;charset=utf-8
 */
@RequestMapping(value="/handle02",method=RequestMethod.POST)
public String handle02(){
    System.out.println("handle02...");
    return "success";
}

/**
 *
 * @return
 */
@RequestMapping(value="/handle03",params={"username!=123","pwd","!age"})
public String handle03(){
    System.out.println("handle03....");
    return "success";
}

/**
 * User-Agent：浏览器信息；
 * 让火狐能访问，让谷歌不能访问
 *
 * 谷歌：
 * User-Agent:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/58.0.3029.110 Safari/537.36
 * 火狐；
 * User-Agent   Mozilla/5.0 (Windows NT 6.3; WOW64; rv:34.0) Gecko/20100101 Firefox/34.0
 * @return
 *
 */
@RequestMapping(value="/handle04",headers={"User-Agent=Mozilla/5.0 (Windows NT 6.3; WOW64;
rv:34.0) Gecko/20100101 Firefox/34.0"})
public String handle04(){
    System.out.println("handle04....");
    return "success";
}


/**
 * @RequestMapping模糊匹配功能(Ant风格)
 *
 * @author lfy
 *
 * URL地址可以写模糊的通配符：
 *  ?：能替代任意一个字符
 *  *：能替代任意多个字符，和一层路径
 *  **：能替代多层路径
 *
 */
@Controller
public class RequestMappingTest {
    @RequestMapping("/antTest01")
    public String antTest01(){
        System.out.println("antTest01...");
        return "success";
    }

    /**
     * ?匹配一个字符,0个多个都不行;
     *      模糊和精确多个匹配情况下，精确优先
     *
     * @return
     */
    @RequestMapping("/antTest0?")
    public String antTest02(){
        System.out.println("antTest02...");
        return "success";
    }

    /**
     *   *匹配任意多个字符
     * @return
     */
    @RequestMapping("/antTest0*")
    public String antTest03(){
        System.out.println("antTest03...");
        return "success";
    }

    /**
     *  *: 匹配一层路径
     * @return
     */
    @RequestMapping("/a/*/antTest01")
    public String antTest04(){
        System.out.println("antTest04...");
        return "success";
    }

    @RequestMapping("/a/**/antTest01")
    public String antTest05(){
        System.out.println("antTest05...");
        return "success";
    }

    //路径上可以有占位符:   占位符 语法就是可以在任意路径的地方写一个{变量名}
    //@PathVariable可以使用这个注解获取
    //   /user/admin    /user/rooo
    // 路径上的占位符必须与整个请求路径层数相同 如果为/user/rooo/asd就不行
    @RequestMapping("/user/{id}")
    public String pathVariableTest(@PathVariable("id")String id){
        System.out.println("路径上的占位符的值"+id);
        return "success";
    }
    //路径上可以有占位符可以多层使用
    @RequestMapping("/{user}/{id}")
    public String pathVariableTest(@PathVariable("user")String user,@PathVariable("id")String id){
        System.out.println("路径上的占位符的值"+id+"---"+user);
        return "success";
    }
}
```

6.REST风格编程

Rest：系统希望以非常简洁的URL地址来发请求； 怎样表示对一个资源的增删改查用请求方式来区分

/getBook?id=1 ： 查询图书 /deleteBook?id=1： 删除1号图书 /updateBook?id=1:更新1号图书 /addBook ： 添加图书 Rest推荐； url地址这么起名； /资源名/资源标识符 /book/1 ： GET-----查询1号图书 /book/1 :PUT------更新1号图书 /book/1 :DELETE-----删除1号图书 /book :POST-----添加图书 系统的URL地址就这么来设计即可； 简洁的URL提交请求，以请求方式区分对资源操作； 问题：从页面上只能发起两种请求，GET、POST； 其他的请求方式没法使用,可以使用如下方法解决；

从页面发起PUT、DELETE形式的请求?Spring提供了对Rest风格的支持 1）、SpringMVC中有一个Filter；他可以把普通的请求转化为规定形式的请求；配置这个filter;

```
在web.xml中配置
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

查看HiddenHttpMethodFilter的源码可知原理:

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
        throws ServletException, IOException {

    /** Default method parameter: {@code _method} */
    public static final String DEFAULT_METHOD_PARAM = "_method";

    private String methodParam = DEFAULT_METHOD_PARAM;


    /**
     * Set the parameter name to look for HTTP methods.
     * @see #DEFAULT_METHOD_PARAM
     */
    public void setMethodParam(String methodParam) {
        Assert.hasText(methodParam, "'methodParam' must not be empty");
        this.methodParam = methodParam;
    }

    HttpServletRequest requestToUse = request;

    if ("POST".equals(request.getMethod()) &&
request.getAttribute(WebUtils.ERROR_EXCEPTION_ATTRIBUTE) == null) {
        //获取表单上_method带来的值 (delete\put)
        String paramValue = request.getParameter(this.methodParam);
        if (StringUtils.hasLength(paramValue)) {
            //转为PUT、DELETE
            requestToUse = new HttpMethodRequestWrapper(request, paramValue);
        }
    }

    filterChain.doFilter(requestToUse, response);
}
private static class HttpMethodRequestWrapper extends HttpServletRequestWrapper {

    private final String method;

    public HttpMethodRequestWrapper(HttpServletRequest request, String method) {
        super(request);
        this.method = method.toUpperCase(Locale.ENGLISH);
    }
    //重写getMethod方法  这个method就是你你想转的method (PUT、DELETE)
    @Override
    public String getMethod() {
        return this.method;
    }
}
```

处理程序:

```
@Controller
public class BookController {
    /**
     * 处理查询图书请求
     * @param id
     * @return
     */
    @RequestMapping(value="/book/{bid}",method=RequestMethod.GET)
    public String getBook(@PathVariable("bid")Integer id) {
        System.out.println("查询到了"+id+"号图书");
        return "success";
    }

    /**
     * 图书删除
     * @param id
     * @return
     */
    @RequestMapping(value="/book/{bid}",method=RequestMethod.DELETE)
    public String deleteBook(@PathVariable("bid")Integer id) {
        System.out.println("删除了"+id+"号图书");
        return "success";
    }

    /**
     * 图书更新
     * @return
     */
    @RequestMapping(value="/book/{bid}",method=RequestMethod.PUT)
    public String updateBook(@PathVariable("bid")Integer id) {
        System.out.println("更新了"+id+"号图书");
        return "success";
    }

    @RequestMapping(value="/book",method=RequestMethod.POST)
    public String addBook() {
        System.out.println("添加了新的图书");
        return "success";
    }
}
```

2）、如何发其他形式请求？ 按照以下要求；1、创建一个post类型的表单 2、表单项中携带一个_*method的参数，3、这个*method的值就是DELETE、PUT

```
<a href="book/1">查询图书</a><br/>
<form action="book" method="post">
    <input type="submit" value="添加1号图书"/>
</form><br/>
<!-- 发送DELETE请求 -->
<form action="book/1" method="post">
    <!--注:这里必须是_method 否则HiddenHttpMethodFilter这个过滤器无法获取到你想转换的method的值的 -->
    <input name="_method" value="delete"/>
    <input type="submit" value="删除1号图书"/>
</form><br/>
<!-- 发送PUT请求 -->
<form action="book/1" method="post">
    <input name="_method" value="put"/>
    <input type="submit" value="更新1号图书"/>
</form><br/>
```

3.springMvc获取请求参数

```
@Controller
public class HelloController {

    /**
     * servlet直接使用request.getParameter("")就可获得请求参数
     * @return
     */
    @RequestMapping("/hello")
    public String handle01() {
        System.out.println("handle01...");
        return "success";
    }

    /**
     * SpringMVC如何获取请求带来的各种信息 默认方式获取请求参数:  直接给方法入参上写一个和请求参数名相同的变量。这个变量就来接收请求参数的值;
     * 带: 有值, 没带: null;
     *
     * @RequestParam: 获取请求参数的; 参数默认是必须带的;
     *              required:是否请求参数必须带? ture是必须带,false是可以不带
     *              defaultValue:如果你没带, 那么使用这个默认值
     * @RequestParam("user") 相当于String username username =
     *                             request.getParameter("user")
     *
     *
     * @RequestParam("user")
     * @PathVariable("user")
     *                       /book/【{user}pathvariable】?【user=admin(requestparam)
     *                       】
     *
     *                       value:指定要获取的参数的key required:这个参数是否必须的
     *                       defaultValue:默认值。没带默认是null;
     *
     *
     * @RequestHeader: 获取请求头中某个key的值;  request.getHeader("User-Agent");
     * @RequestHeader("User-Agent")String userAgent userAgent =
     *                                   request.getHeader("User-Agent")
     *                                   如果请求头中没有这个值就会报错;  value() required()
     *                                   defaultValue()
     *
     * @CookieValue: 获取某个cookie的值;  以前的操作获取某个cookie;  Cookie[] cookies =
     *                            request.getCookies(); for(Cookie c:cookies){
     *                            if(c.getName().equals("JSESSIONID")){ String
     *                            cv = c.getValue(); } }
     * value()
     * required()
     * defaultValue()
     */
    @RequestMapping("/handle01")
    public String handle02(
            @RequestParam(value = "user", required = false, defaultValue = "你没带") String
username,
            @RequestHeader(value = "AHAHA", required = false, defaultValue = "她也没带") String
userAgent,
            @CookieValue(value="JSESSIONID",required=false)String jid) {
        System.out.println("这个变量的值: " + username);
        System.out.println("请求头中浏览器的信息: " + userAgent);
        System.out.println("cookie中的jid的值"+jid);
        return "success";
    }

    /**
     * 如果我们的请求参数是一个POJO;
     * SpringMVC会自动的为这个POJO进行赋值?
     * 1)、将POJO中的每一个属性，从request参数中尝试获取出来，并封装即可;
     * 2)、还可以级联封装; 属性的属性
     * 3)、请求参数的参数名和对象中的属性名一一对应就行
     *
     *
     * 提交的数据可能有乱码:
     * 请求乱码:
     *         GET请求: 改server.xml; 在8080端口处URIEncoding="UTF-8"
     *         POST请求:
     *             在第一次获取请求参数之前设置
     *             request.setCharacterEncoding("UTF-8");
     *             自己写一个filter; SpringMVC有这个filter
     *
     * 响应乱码:
     *         response.setContentType("text/html;charset=utf-8")
     * @param book
     * @return
     */
    @RequestMapping("/book")
    public String addBook(Book book){
        System.out.println("我要保存的图书: "+book);
        return "success";
    }

    /**
     * SpringMVC可以直接在参数上写原生API;
     *
     * HttpServletRequest
     * HttpServletResponse
     * HttpSession
     *
     *
     * java.security.Principal
     * Locale: 国际化有关的区域信息对象
     * InputStream:
     *          ServletInputStream inputStream = request.getInputStream();
     * OutputStream:
     *          ServletOutputStream outputStream = response.getOutputStream();
     * Reader:
     *          BufferedReader reader = request.getReader();
     * Writer:
     *          PrintWriter writer = response.getWriter();
     *
     * @throws IOException

     *
     *
     */
    @RequestMapping("/handle03")
    public String handle03(HttpSession session,
            HttpServletRequest request,HttpServletResponse response) throws IOException {
        request.setAttribute("reqParam", "我是请求域中的");
        session.setAttribute("sessionParam", "额我是Session域中的");

        return "success";
    }

}
```

POST请求乱码的解决方法,自己写一个filter,也可以使用SpringMVC已经写filter

```
在web.xml中配置这个过滤器
注意:字符编码filter一定要在其他的filter之前, 否则其他的filter可能已经先拿出了参数,那么此时再去转化参数已经来不及了
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <!-- 设置编码的值为utf-8编码 -->
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
      <!-- 设置响应乱码为true -->
```

```
        <param-name>forceResponseEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

CharacterEncodingFilter的源码

```
@Override
protected void doFilterInternal(
        HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {

    String encoding = getEncoding();
    if (encoding != null) {
        if (isForceRequestEncoding() || request.getCharacterEncoding() == null) {
            //实际上也是调用了request.setCharacterEncoding("UTF-8");这个方法
            request.setCharacterEncoding(encoding);
        }
        if (isForceResponseEncoding()) {
            //如果forceResponseEncoding为ture则设置响应乱码的类型也是encoding
            response.setCharacterEncoding(encoding);
        }
    }
    filterChain.doFilter(request, response);
}
```

4.数据输出

数据输出：如何将数据带给页面；

```
/**
 * SpringMVC除过在方法上传入原生的request和session外还能怎么样把数据带给页面
 *
 * 1）、可以在方法处传入Map、或者Model或者ModelMap。
 *       给这些参数里面保存的所有数据都会放在请求域中。可以在页面获取
 *    关系：
 *       Map，Model，ModelMap：最终都是BindingAwareModelMap在工作；
 *       相当于给BindingAwareModelMap中保存的东西都会被放在请求域中；
 *
 *       Map(interface(jdk))      Model(interface(spring))
 *           ||                           //
 *           ||                          //
 *           \/                         //
 *       ModelMap(clas)                //
 *                   \\               //
 *                    \\             //
 *                   ExtendedModelMap
 *                           ||
 *                           \/
```

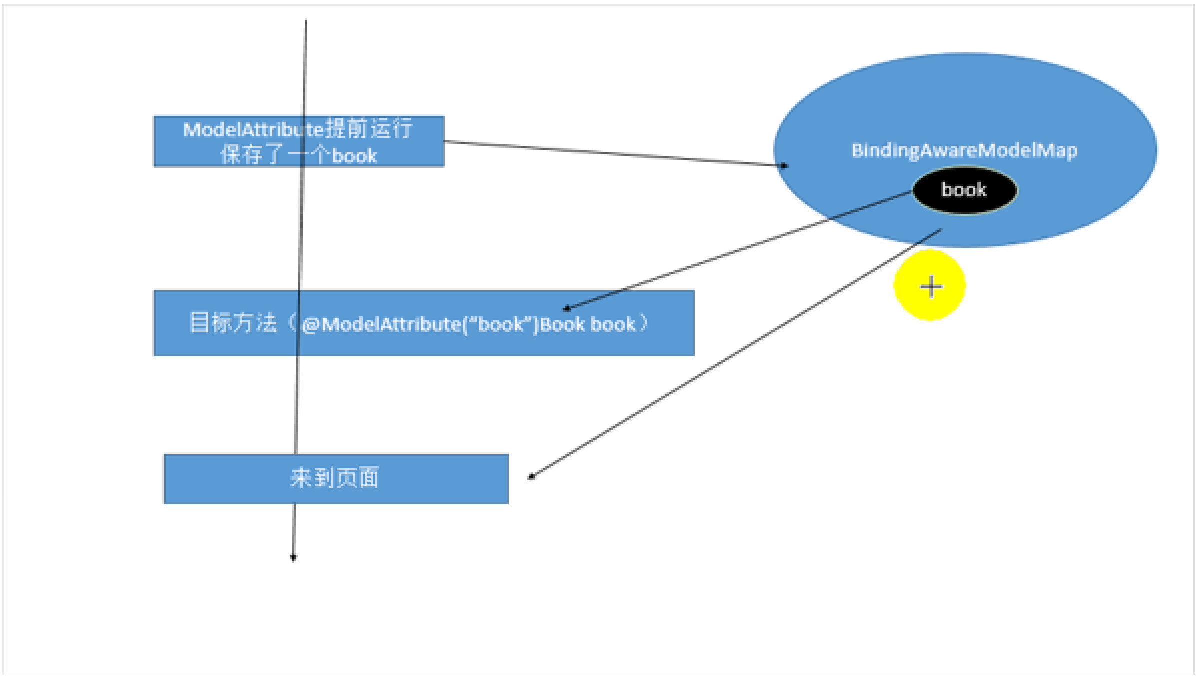
```java
 *                  BindingAwareModelMap
 *
 * 2) 、方法的返回值可以变为ModelAndView类型;
 *          既包含(View)视图信息（页面地址）也包含(Model)模型数据（给页面带的数据）;
 *          而且数据是放在请求域中;（SpringMvc选择将给页面的返回值放进Request域)
 *          可以使用的域数据类型:request、session、application;
 *
 *  3)、也可以将参数放在原声API中,比如:Request,Session等中
 *
 *
 */
@Controller
public class OutputController {

    @RequestMapping("/handle01")
    public String handle01(Map<String, Object> map){
        map.put("msg", "你好");
        System.out.println("map的类型: "+map.getClass());
        return "success";
    }

    /**
     * Model: 一个接口
     * @param model
     * @return
     */
    @RequestMapping("/handle02")
    public String handle02(Model model){
        model.addAttribute("msg", "你好啊! ");
        System.out.println("model的类型: "+model.getClass());
        return "success";
    }

    @RequestMapping("/handle03")
    public String handle03(ModelMap modelMap){
        modelMap.addAttribute("msg", "你好吗? ");
        System.out.println("modelmap的类型: "+modelMap.getClass());
        return "success";
    }

    /**
     * 返回值是ModelAndView;可以为页面携带数据
     * @return
     */
    @RequestMapping("/handle04")
    public ModelAndView handle04(){
        //之前的返回值我们就叫视图名; 视图名视图解析器是会帮我们最终拼串得到页面的真实地址;
        //ModelAndView mv = new ModelAndView("success");
        ModelAndView mv = new ModelAndView();
        mv.setViewName("success");
        mv.addObject("msg", "你好哦! ");
        return mv;

    }

    /**
    *  使用原生的API给页面返回值
```

```
     * HttpServletRequest
     * HttpServletResponse
     * HttpSession
     *
     *
     * java.security.Principal
     * Locale: 国际化有关的区域信息对象
     * InputStream:
     *         ServletInputStream inputStream = request.getInputStream();
     * OutputStream:
     *         ServletOutputStream outputStream = response.getOutputStream();
     * Reader:
     *         BufferedReader reader = request.getReader();
     * Writer:
     *         PrintWriter writer = response.getWriter();
     */
    @RequestMapping("/handle05")
    public String handle05(Request request,Session session){

        request.setAttribute("msg","你好好啊! ");
        session.setAttribute("msgSession","我也好啊! ");
        return "success";
    }
}
```

@ModelAttribute注解

```
/**
 * 测试ModelAttribute注解;
 * 使用场景: 书城的图书修改为例;
 * 1) 页面端;
 *      显示要修改的图书的信息, 图书的所有字段都在
 * 2) servlet收到修改请求, 调用dao;
 *      String sql="update bs_book set title=?,
 *                  author=?,price=?,
 *                  sales=?,stock=?,img_path=?
 *              where id=?";
 * 3) 实际场景?
 *      并不是全字段修改; 只会修改部分字段, 以修改用户信息为例;
 *      username  password  address;而实际业务中username是不会修改的而只会修改其他例如密码,地址等信息
 *      1) 、不修改的字段可以在页面进行展示但是不要提供修改输入框;
 *      2) 、为了简单, Controller直接在参数位置来写Book对象
 *      3) 、SpringMVC为我们自动封装book; (没有带的值是null)
 *      4) 、如果接下来调用了一个全字段更新的dao操作; 会将其他的字段可能变为null;
 *          sql = "update bs_book set"
 *          if(book.getBookName()){
 *              sql +="bookName=?,"
 *          }
 *          if(book.getPrice()){
 *              sql +="price=?"
 *          }
 *
 * 4) 、如何能保证全字段更新的时候, 只更新了页面携带的数据;
 *      1) 、修改dao; 代价大?
 *      2) 、Book对象是如何封装的?
 *          1) 、SpringMVC创建一个book对象, 每个属性都有默认值, bookName就是null;
 *              1、让SpringMVC别创建book对象，直接从数据库中先取出一个id=100的book对象的信息
 *              2、Book [id=100, bookName=西游记, author=张三, stock=12, sales=32, price=98.98]
 *
 *          2）、将请求中所有与book对应的属性一一设置过来；
 *              3、使用刚才从数据库取出的book对象，给它 的里面设置值； （请求参数带了哪些值就覆盖之前的值）
 *              4、带了的字段就改为携带的值，没带的字段就保持之前的值
 *          3）、调用全字段更新就有问题；
 *              5、将之前从数据库中查到的对象，并且封装了请求参数的对象。进行保存；
 *
 */
@Controller
public class ModelAttributeTestController {

    private Object o1;
    private Object o2;

    private Object b1;
    private Object b2;

    //bookDao.update(book);
    //Book [id=100, bookName=null, author=张三, stock=12, sales=32, price=98.98]
    /**
     *      String sql="update bs_book set bookName=?,
                    author=?,price=?,
                    sales=?,stock=?,img_path=?
                where id=?";
     */
    /**
     * 可以告诉SpringMVC不要new这个book了我刚才保存了一个book；
     * 哪个就是从数据库中查询出来的；用我这个book?@ModelAttribute("haha")
     *
     *
     * 同都是BindingAwareModelMap
     * @param book
     * @return
     */
    @RequestMapping("/updateBook")
    public String updateBook(@ModelAttribute("haha")Book book,Map<String, Object> model){
        o2 = model;
        b2  = book;
        Object haha = model.get("haha");
        //System.out.println("传入的model: "+model.getClass());
        System.out.println("o1==o2?"+(o1 == o2)); //true
        System.out.println("b1==b2?"+(b1 == b2)+"-->"+(b2 == haha)); //true

        System.out.println("页面要提交过来的图书信息: "+book);
        return "success";
    }

    /**
     * 1）、SpringMVC要封装请求参数的Book对象不应该是自己new出来的。
     *      而应该是【从数据库中】拿到的准备好的对象
     * 2）、再来使用这个对象封装请求参数
     *
     * @ModelAttribute:
     *      参数：取出刚才保存的数据
     *      方法位置：这个方法就会提前于目标方法先运行；
```

```
      *           1)我们可以在这里提前查出数据库中图书的信息
      *           2)将这个图书信息保存起来（方便下一个方法还能使用）
      *
      * 参数的map: BindingAwareModelMap
      */
    @ModelAttribute
    public void hahaMyModelAttribute(Map<String, Object> map){

        Book book = new Book(100, "西游记", "吴承恩", 98, 10, 98.98);
        System.out.println("数据库中查到的图书信息是："+book);
        map.put("haha", book);
        b1 = book;
        o1 = map;
        System.out.println("modelAttribute方法...查询了图书并给你保存起来了...他用的map的类
型："+map.getClass());
    }
}
```



注：@ModelAttribute的整个关系运行图

5.视图解析

SpringMVC视图解析；

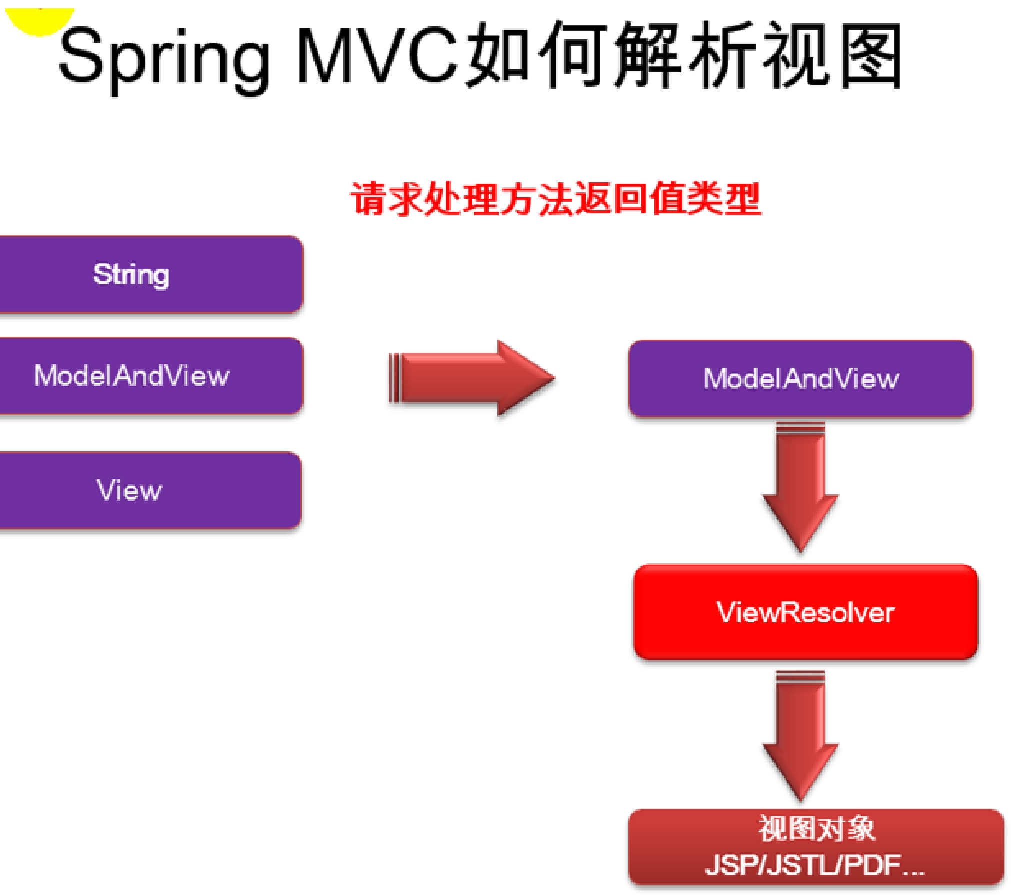
1、方法执行后的返回值会作为页面地址参考，转发或者重定向到页面

2、视图解析器可能会进行页面地址的拼串；

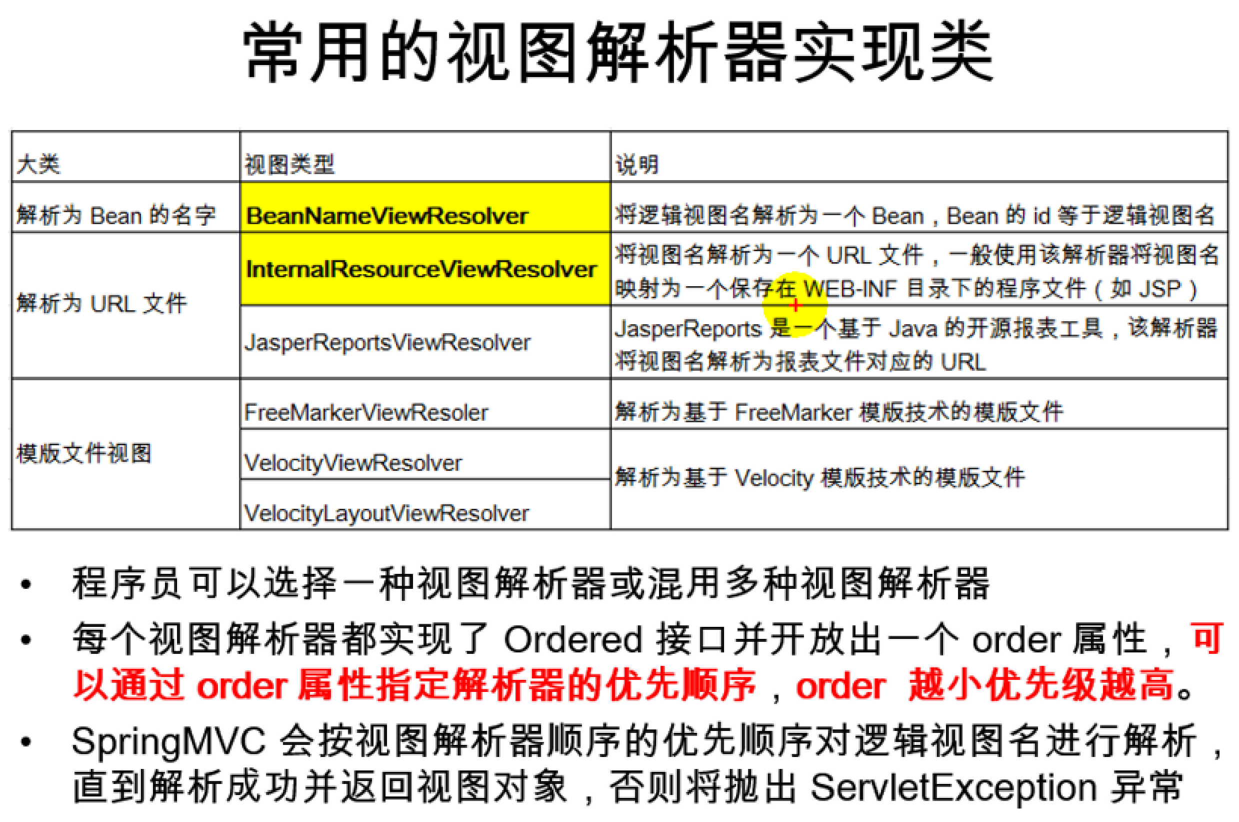
```
//去往视图页面的基本使用方法
@Controller
public class HelloController {

    @RequestMapping("/hello")
    public String hello(){
        //          /WEB-INF/pages/hello.jsp    /hello.jsp
        //相对路径
        return "../../hello";
    }

    /**
     *  forward:转发到一个页面
     *  /hello.jsp: 转发当前项目下的hello;
     *
     *  一定加上/, 如果不加/就是相对路径。容易出问题;
     *  forward:/hello.jsp
     *  forward:前缀的转发, 不会由我们配置的视图解析器拼串
     *
     * @return
     */
    @RequestMapping("/handle01")
    public String handle01(){
        System.out.println("handle01");
        return "forward:/hello.jsp";
    }

    @RequestMapping("/handle02")
    public String handle02(){
        System.out.println("handle02");
        return "forward:/handle01";
    }

    /**
     * 重定向到hello.jsp页面
     * 有前缀的转发和重定向操作, 配置的视图解析器就不会进行拼串;
     *
     * 转发    forward:转发的路径
     * 重定向(重定向路径由浏览器解析) redirect:重定向的路径
     *      /hello.jsp:代表就是从当前项目下开始; SpringMVC会为路径自动的拼接上项目名
     *
     *      原生的Servlet重定向/路径需要加上项目名才能成功   但是springMvc就不用,它会自动加的
     *      response.sendRedirect("/hello.jsp")
     * @return
     */
    @RequestMapping("/handle03")
    public String handle03(){
        System.out.println("handle03....");
        return "redirect:/hello.jsp";
    }

    @RequestMapping("/handle04")
    public String handle04(){
        System.out.println("handle04...");
        return "redirect:/handle03";
    }

//  @RequestMapping("/toLoginPage")
//  public String toLogin(){
//      //return "forward:/WEB-INF/pages/login.jsp";
//      return "login";
//  }
```

```
}
```

springMvc视图解析的基本流程如下图:



视图的作用就是渲染模型数据，将模型数据里的数据以某种形式呈现给用户。我们可以呈现页面的形式，我也可以生成一个excel，pdf等 还可以生成一个链接的形式给你 对此视图这个东西不要局限的理解为就是页面 这违背springMvc的设计原则。

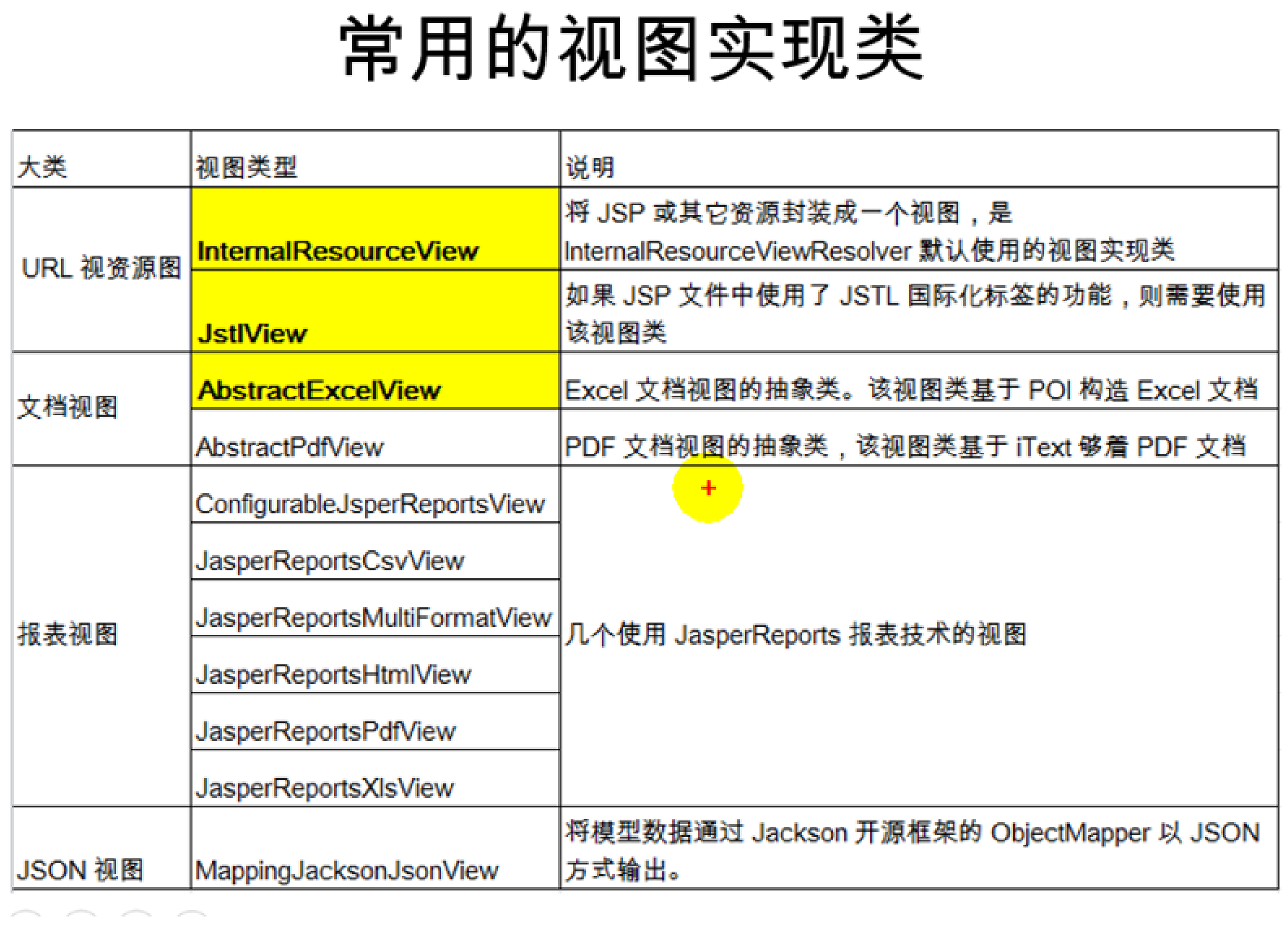


常用的视图解析器实现类

大类	视图类型	说明
解析为 Bean 的名字	BeanNameViewResolver	将逻辑视图名解析为一个 Bean，Bean 的 id 等于逻辑视图名
解析为 URL 文件	InternalResourceViewResolver	将视图名解析为一个 URL 文件，一般使用该解析器将视图名映射为一个保存在 WEB-INF 目录下的程序文件（如 JSP）
	JasperReportsViewResolver	JasperReports 是一个基于 Java 的开源报表工具，该解析器将视图名解析为报表文件对应的 URL
模版文件视图	FreeMarkerViewResoler	解析为基于 FreeMarker 模版技术的模版文件
	VelocityViewResolver	解析为基于 Velocity 模版技术的模版文件
	VelocityLayoutViewResolver	

- 程序员可以选择一种视图解析器或混用多种视图解析器
- 每个视图解析器都实现了 Ordered 接口并开放出一个 order 属性，**可以通过 order 属性指定解析器的优先顺序，order 越小优先级越高。**
- SpringMVC 会按视图解析器顺序的优先顺序对逻辑视图名进行解析，直到解析成功并返回视图对象，否则将抛出 ServletException 异常

注:常用的视图解析器类及作用



常用的视图实现类

大类	视图类型	说明
URL 视资源图	InternalResourceView	将 JSP 或其它资源封装成一个视图，是 InternalResourceViewResolver 默认使用的视图实现类
	JstlView	如果 JSP 文件中使用了 JSTL 国际化标签的功能，则需要使用该视图类
文档视图	AbstractExcelView	Excel 文档视图的抽象类。该视图类基于 POI 构造 Excel 文档
	AbstractPdfView	PDF 文档视图的抽象类，该视图类基于 iText 够着 PDF 文档
报表视图	ConfigurableJsperReportsView	几个使用 JasperReports 报表技术的视图
	JasperReportsCsvView	
	JasperReportsMultiFormatView	
	JasperReportsHtmlView	
	JasperReportsPdfView	
	JasperReportsXlsView	
JSON 视图	MappingJacksonJsonView	将模型数据通过 Jackson 开源框架的 ObjectMapper 以 JSON 方式输出。

可以在配置视图解析器类型的时候强制指定其创建某种视图，配置如下：在spring-mvc.xml配置文件中

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value="/WEB-INF/views/"></property>
    <property name = "suffix" value = ".jsp"></property>
    <!-- 强制指定其解析的视图为JstlView -->
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"></property>
</bean>
```

说到了Jstl标签就顺便说一下springMvc管理国际化

1.javaWeb国际化步骤;

1.得到一个Locale对象;

2.使用ResourceBundle绑定国际化资源文件;

3.使用ResourceBundle.getString("key");获取到国际化配置文件中的值;

4.web页面的国际化，fmt标签库来做;

fmt:setLocale

<fmt:setBundle >

fmt:message

2.有了JstlView以后;

1.让Spring管理国际化资源就行,在spring-mvc.xml中配置

```
<!--让SpringMVC管理国际化资源文件;配置一个资源文件管理器 ID名必须是messageSource -->
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
    <!--  basename指定基础名-->
    <property name="basename" value="i18n"></property>
</bean>
```

```
//ID名必须是messageSource的原因
//在IOC容器的刷新时会刷新国际化相关东西 在AbstractApplicationContext文件中
protected void initMessageSource() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (beanFactory.containsLocalBean(MESSAGE_SOURCE_BEAN_NAME)) {
        //public static final String MESSAGE_SOURCE_BEAN_NAME = "messageSource";
        //容器刷新时会去容器中找ID名为messageSource的bean 所以你的你的国际化管理器必须叫messageSource
        this.messageSource = beanFactory.getBean(MESSAGE_SOURCE_BEAN_NAME, MessageSource.class);
        // Make MessageSource aware of parent MessageSource.
        if (this.parent != null && this.messageSource instanceof HierarchicalMessageSource) {
            HierarchicalMessageSource hms = (HierarchicalMessageSource) this.messageSource;
            if (hms.getParentMessageSource() == null) {
                // Only set parent context as parent MessageSource if no parent MessageSource
                // registered already.
                hms.setParentMessageSource(getInternalParentMessageSource());
            }
        }
        if (logger.isDebugEnabled()) {
            logger.debug("Using MessageSource [" + this.messageSource + "]");
        }
    }
    else {
        // Use empty MessageSource to be able to accept getMessage calls.
```

```
            DelegatingMessageSource dms = new DelegatingMessageSource();
            dms.setParentMessageSource(getInternalParentMessageSource());
            this.messageSource = dms;
            beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME, this.messageSource);
            if (logger.isDebugEnabled()) {
                logger.debug("Unable to locate MessageSource with name '" + MESSAGE_SOURCE_BEAN_NAME +
                        "': using default [" + this.messageSource + "]");
            }
        }
    }
```

2.直接去页面使用fmt:message；

一定要过SpringMVC的视图解析流程，人家会创建一个jstlView帮你快速国际化；

也不能写forward:前缀这么开头

扩展：

1.加深视图解析器和视图对象；

2.视图解析器根据方法的返回值得到视图对象；

3.多个视图解析器都会尝试能否得到视图对象；（抽出部分源码,具体源码分析中有）

```
for (ViewResolver viewResolver : this.viewResolvers) {
   //viewResolver视图解析器根据方法的返回值，得到一个View对象；
   View view = viewResolver.resolveViewName(viewName, locale);
   if (view != null) {
       return view;
   }
}
```

4.视图对象不同就可以具有不同功能；

自定义视图和视图解析器:

1.自定义视图和视图解析器的步骤；

1.编写自定义的视图解析器，和视图实现类

2.视图解析器必须放在ioc容器中，让其工作，能创建出我们的自定义视图对象；

2.自定视图和视图解析器工作原理

1.让我们的视图解析器工作；

2.得到我们的视图对象

3.我们的视图对象自定义渲染逻辑

```
//控制器
@Controller
public class MyViewResovlerController {

    @RequestMapping("/handleplus")
    public String handleplus(Model model){
        //myView:/aa  myView:/bb
        //forward:/login.jsp
        List<String> vname = new ArrayList<String>();
        List<String> imgname = new ArrayList<String>();
        vname.add("AA");
        vname.add("BB");
        imgname.add("CC");

        model.addAttribute("video", vname);
        model.addAttribute("imgs", imgname);

        return "myView:/aa";
    }

}
//自定义视图解析器
public class MyMeiNVViewResolver implements ViewResolver,Ordered{

    private Integer order = 0;

    @Override
    public View resolveViewName(String viewName, Locale locale)
            throws Exception {
        // TODO Auto-generated method stub
        //根据视图名返回视图对象
        /**
         *  //myView:/aa  myView:/bb
            forward:/login.jsp
         */
        if(viewName.startsWith("meinv:")){
            return new MyView();
        }else{
            //如果不能处理返回null即可 因为这只是我们自定义的视图并不能解析所有视图 所以无法解析时就返回null 让视图解析器再去循环请别的视图解析器继续继续 源码分析哪里有讲
            return null;
        }
    }


    @Override
    public int getOrder() {
        return order;
    }

    //改变视图解析器的优先级 可以让自定义的优先级更高  尽量让我们自定义的视图解析器先解析  因为默认的InternalResourceViewResolver这个视图解析器是个万能型的解析器 只要解析不了它就拼串 不会返回null的(所以springMvc将他自己的这个解析器优先级调的最低 private int order = Ordered.LOWEST_PRECEDENCE;最低级别为Integer.MAX_VALUE数字越大级别越小)  所以轮不到自定义的解析器
    public void setOrder(Integer order){
        this.order = order;
    }

}
//自定义视图
public class MyView implements View{

    /**
     * 返回的数据的内容类型
     */
    @Override
    public String getContentType() {
        return "text/html";
    }

    @Override
    public void render(Map<String, ?> model, HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        System.out.println("之前保存的数据："+model);
        response.setContentType("text/html");
        List<String> vn = (List<String>) model.get("video");
        response.getWriter().write("<h1>即将展现精彩内容</h1>");
        for (String string : vn) {
            response.getWriter().write("<a>下载"+string+".avi</a><br/>");
        }
    }

}
//最后需要写配置文件将自定义视图解析器加入到IOC容器中
```

6.数据转换 & 数据格式化 & 数据校验

SpringMVC封装自定义类型对象的时候？ javaBean要和页面提交的数据进行一一绑定？是的 1.页面提交的所有数据都是字符串？是的 http协议(超文本传输协议) 传输的自然都是字符串 2.Integer age,Date birth; 而bean中可能就有日期,数字等 所以需要进行转换

```
employName=zhangsan&age=18&gender=1
String age = request.getParameter("age");
```

牵扯到以下操作； 1.数据绑定期间的数据类型转换？String--Integer String--Boolean,xxx 2.数据绑定期间的数据格式化问题？比如提交的日期进行转换 birth=2017-12-15----->Date 2017/12/15 2017.12.15 2017-12-15 3.数据校验？ 我们提交的数据必须是合法的？ 前端校验：js+正则表达式；(实际前端的校验并不是我们想象的那么安全 因为每一个浏览器都可以禁用js) 后端校验：重要数据也是必须的； 1.校验成功！数据合法 2.校验失败？保存校验结果

上述的问题springMvc都是交给DataBinder这个类来做

WebDataBinder(DataBinder的接口)：数据绑定器负责数据绑定工作；

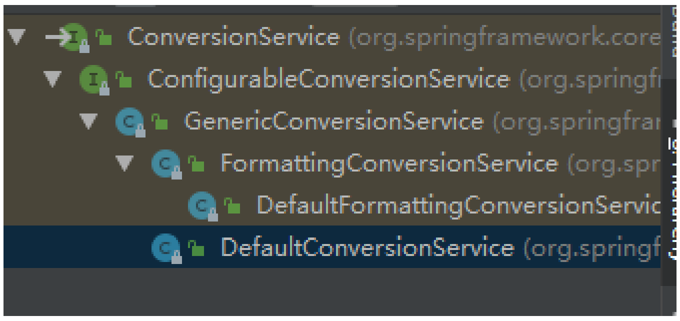
数据绑定期间产生的类型转换、格式化、数据校验等问题；

```
//DataBinder类中的重要字段
public class DataBinder implements PropertyEditorRegistry, TypeConverter {

    private final Object target;

    private final String objectName;

    //bindingResult负责保存以及解析数据绑定期间数据校验产生的错误
    private AbstractPropertyBindingResult bindingResult;

    private SimpleTypeConverter typeConverter;

    private boolean ignoreUnknownFields = true;
    private boolean ignoreInvalidFields = false;

    private boolean autoGrowNestedPaths = true;

    private int autoGrowCollectionLimit = DEFAULT_AUTO_GROW_COLLECTION_LIMIT;

    private String[] allowedFields;

    private String[] disallowedFields;

    private String[] requiredFields;

    //ConversionService组件：负责数据类型的转换以及格式化功能；
    private ConversionService conversionService;

    private MessageCodesResolver messageCodesResolver;

    private BindingErrorProcessor bindingErrorProcessor = new DefaultBindingErrorProcessor();

    //validators负责数据校验工作
    private final List<Validator> validators = new ArrayList<Validator>();
}
```

ConversionService组件：负责数据类型的转换以及格式化功能；

ConversionService中有非常多的converter

不同类型的转换和格式化用它自己的converter



```
ConversionService (org.springframework.core
  ConfigurableConversionService (org.springfr
    GenericConversionService (org.springfrar
      FormattingConversionService (org.spr
        DefaultFormattingConversionServic
      DefaultConversionService (org.springf
```

注:ConversionService接口的主要实现类

```
//ConversionService接口
public interface ConversionService {

    boolean canConvert(Class<?> sourceType, Class<?> targetType);

    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

    <T> T convert(Object source, Class<T> targetType);

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}
//以DefaultConversionService为例简要的分析下其实现方式
public class DefaultConversionService extends GenericConversionService {
    private static final boolean javaUtilOptionalClassAvailable =
            ClassUtils.isPresent("java.util.Optional",
DefaultConversionService.class.getClassLoader());


    private static final boolean jsr310Available =
            ClassUtils.isPresent("java.time.ZoneId",
DefaultConversionService.class.getClassLoader());


    private static final boolean streamAvailable = ClassUtils.isPresent(
            "java.util.stream.Stream", DefaultConversionService.class.getClassLoader());

    private static volatile DefaultConversionService sharedInstance;

    public DefaultConversionService() {
        //构造方法 其大量的为其注册Converter
        addDefaultConverters(this);
    }



    public static ConversionService getSharedInstance() {
        if (sharedInstance == null) {
            synchronized (DefaultConversionService.class) {
                if (sharedInstance == null) {
                    sharedInstance = new DefaultConversionService();
                }
            }
        }
        return sharedInstance;
    }


    public static void addDefaultConverters(ConverterRegistry converterRegistry) {
        //这是添加基础类型的一些转化器
        addScalarConverters(converterRegistry);
        //这是添加集合类型的一些转化器
        addCollectionConverters(converterRegistry);

        converterRegistry.addConverter(new ByteBufferConverter((ConversionService)
converterRegistry));
        if (jsr310Available) {
            Jsr310ConverterRegistrar.registerJsr310Converters(converterRegistry);
        }

        converterRegistry.addConverter(new ObjectToObjectConverter());
        converterRegistry.addConverter(new IdToEntityConverter((ConversionService)
converterRegistry));
        converterRegistry.addConverter(new FallbackObjectToStringConverter());
        if (javaUtilOptionalClassAvailable) {
            converterRegistry.addConverter(new ObjectToOptionalConverter((ConversionService)
converterRegistry));
        }
    }
    public static void addCollectionConverters(ConverterRegistry converterRegistry) {
        ConversionService conversionService = (ConversionService) converterRegistry;

        converterRegistry.addConverter(new ArrayToCollectionConverter(conversionService));
        converterRegistry.addConverter(new CollectionToArrayConverter(conversionService));

        converterRegistry.addConverter(new ArrayToArrayConverter(conversionService));
        converterRegistry.addConverter(new CollectionToCollectionConverter(conversionService));
        converterRegistry.addConverter(new MapToMapConverter(conversionService));

        converterRegistry.addConverter(new ArrayToStringConverter(conversionService));
        converterRegistry.addConverter(new StringToArrayConverter(conversionService));

        converterRegistry.addConverter(new ArrayToObjectConverter(conversionService));
        converterRegistry.addConverter(new ObjectToArrayConverter(conversionService));

        converterRegistry.addConverter(new CollectionToStringConverter(conversionService));
        converterRegistry.addConverter(new StringToCollectionConverter(conversionService));

        converterRegistry.addConverter(new CollectionToObjectConverter(conversionService));
        converterRegistry.addConverter(new ObjectToCollectionConverter(conversionService));

        if (streamAvailable) {
            converterRegistry.addConverter(new StreamConverter(conversionService));
        }
    }

    private static void addScalarConverters(ConverterRegistry converterRegistry) {
        converterRegistry.addConverterFactory(new NumberToNumberConverterFactory());

        converterRegistry.addConverterFactory(new StringToNumberConverterFactory());
        converterRegistry.addConverter(Number.class, String.class, new ObjectToStringConverter());

        converterRegistry.addConverter(new StringToCharacterConverter());
        converterRegistry.addConverter(Character.class, String.class, new
ObjectToStringConverter());

        converterRegistry.addConverter(new NumberToCharacterConverter());
        converterRegistry.addConverterFactory(new CharacterToNumberFactory());

        converterRegistry.addConverter(new StringToBooleanConverter());
        converterRegistry.addConverter(Boolean.class, String.class, new
ObjectToStringConverter());

        converterRegistry.addConverterFactory(new StringToEnumConverterFactory());
        converterRegistry.addConverter(new EnumToStringConverter((ConversionService)
converterRegistry));

        converterRegistry.addConverterFactory(new IntegerToEnumConverterFactory());
        converterRegistry.addConverter(new EnumToIntegerConverter((ConversionService)
converterRegistry));

        converterRegistry.addConverter(new StringToLocaleConverter());
        converterRegistry.addConverter(Locale.class, String.class, new ObjectToStringConverter());

        converterRegistry.addConverter(new StringToCharsetConverter());
```

```java
        converterRegistry.addConverter(Charset.class, String.class, new
ObjectToStringConverter());

        converterRegistry.addConverter(new StringToCurrencyConverter());
        converterRegistry.addConverter(Currency.class, String.class, new
ObjectToStringConverter());

        converterRegistry.addConverter(new StringToPropertiesConverter());
        converterRegistry.addConverter(new PropertiesToStringConverter());

        converterRegistry.addConverter(new StringToUUIDConverter());
        converterRegistry.addConverter(UUID.class, String.class, new ObjectToStringConverter());
    }

    private static final class Jsr310ConverterRegistrar {

        public static void registerJsr310Converters(ConverterRegistry converterRegistry) {
            converterRegistry.addConverter(new StringToTimeZoneConverter());
            converterRegistry.addConverter(new ZoneIdToTimeZoneConverter());
            converterRegistry.addConverter(new ZonedDateTimeToCalendarConverter());
        }
    }

}
//ConversionService的来源?来源于FormattingConversionServiceFactoryBean中
```

```java
//converter接口
public interface Converter<S, T> {
    //转化的方法
   T convert(S source);
}
```

```
ConversionService组件中各各样的converter转换器
ConversionService converters =
    @org.springframework.format.annotation.DateTimeFormat java.lang.Long -> java.lang.String:
org.springframework.format.datetime.DateTimeFormatAnnotationFormatterFactory@32abc654,@org.springf
ramework.format.annotation.NumberFormat java.lang.Long -> java.lang.String:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    @org.springframework.format.annotation.DateTimeFormat java.time.LocalDate -> java.lang.String:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.time.LocalDate -> java.lang.String :
org.springframework.format.datetime.standard.TemporalAccessorPrinter@edd23ab
    @org.springframework.format.annotation.DateTimeFormat java.time.LocalDateTime ->
java.lang.String:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.time.LocalDateTime -> java.lang.String :
org.springframework.format.datetime.standard.TemporalAccessorPrinter@4f10103d
    @org.springframework.format.annotation.DateTimeFormat java.time.LocalTime -> java.lang.String:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.time.LocalTime -> java.lang.String :
org.springframework.format.datetime.standard.TemporalAccessorPrinter@2b482406
```

```
    @org.springframework.format.annotation.DateTimeFormat java.time.OffsetDateTime ->
java.lang.String:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.time.OffsetDateTime -> java.lang.String :
org.springframework.format.datetime.standard.TemporalAccessorPrinter@5bff2a4e
    @org.springframework.format.annotation.DateTimeFormat java.time.OffsetTime ->
java.lang.String:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.time.OffsetTime -> java.lang.String :
org.springframework.format.datetime.standard.TemporalAccessorPrinter@220f2940
    @org.springframework.format.annotation.DateTimeFormat java.time.ZonedDateTime ->
java.lang.String:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.time.ZonedDateTime -> java.lang.String :
org.springframework.format.datetime.standard.TemporalAccessorPrinter@174673c
    @org.springframework.format.annotation.DateTimeFormat java.util.Calendar -> java.lang.String:
org.springframework.format.datetime.DateTimeFormatAnnotationFormatterFactory@32abc654
    @org.springframework.format.annotation.DateTimeFormat java.util.Date -> java.lang.String:
org.springframework.format.datetime.DateTimeFormatAnnotationFormatterFactory@32abc654
    @org.springframework.format.annotation.NumberFormat java.lang.Double -> java.lang.String:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    @org.springframework.format.annotation.NumberFormat java.lang.Float -> java.lang.String:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    @org.springframework.format.annotation.NumberFormat java.lang.Integer -> java.lang.String:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    @org.springframework.format.annotation.NumberFormat java.lang.Short -> java.lang.String:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    @org.springframework.format.annotation.NumberFormat java.math.BigDecimal -> java.lang.String:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    @org.springframework.format.annotation.NumberFormat java.math.BigInteger -> java.lang.String:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.Boolean -> java.lang.String :
org.springframework.core.convert.support.ObjectToStringConverter@67fe49bf
    java.lang.Character -> java.lang.Number :
org.springframework.core.convert.support.CharacterToNumberFactory@5184d61b
    java.lang.Character -> java.lang.String :
org.springframework.core.convert.support.ObjectToStringConverter@6a45e172
    java.lang.Enum -> java.lang.String :
org.springframework.core.convert.support.EnumToStringConverter@64048fc3
    java.lang.Long -> java.util.Calendar :
org.springframework.format.datetime.DateFormatterRegistrar$LongToCalendarConverter@37cbfe30
    java.lang.Long -> java.util.Date :
org.springframework.format.datetime.DateFormatterRegistrar$LongToDateConverter@79fc7df4
    java.lang.Number -> java.lang.Character :
org.springframework.core.convert.support.NumberToCharacterConverter@6d7337c1
    java.lang.Number -> java.lang.Number :
org.springframework.core.convert.support.NumberToNumberConverterFactory@6e0b7bd8
    java.lang.Number -> java.lang.String :
org.springframework.core.convert.support.ObjectToStringConverter@1135afc3
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat java.lang.Long:
org.springframework.format.datetime.DateTimeFormatAnnotationFormatterFactory@32abc654,java.lang.St
ring -> @org.springframework.format.annotation.NumberFormat java.lang.Long:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat java.time.LocalDate:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.lang.String -> java.time.LocalDate:
org.springframework.format.datetime.standard.TemporalAccessorParser@6228dd42
```

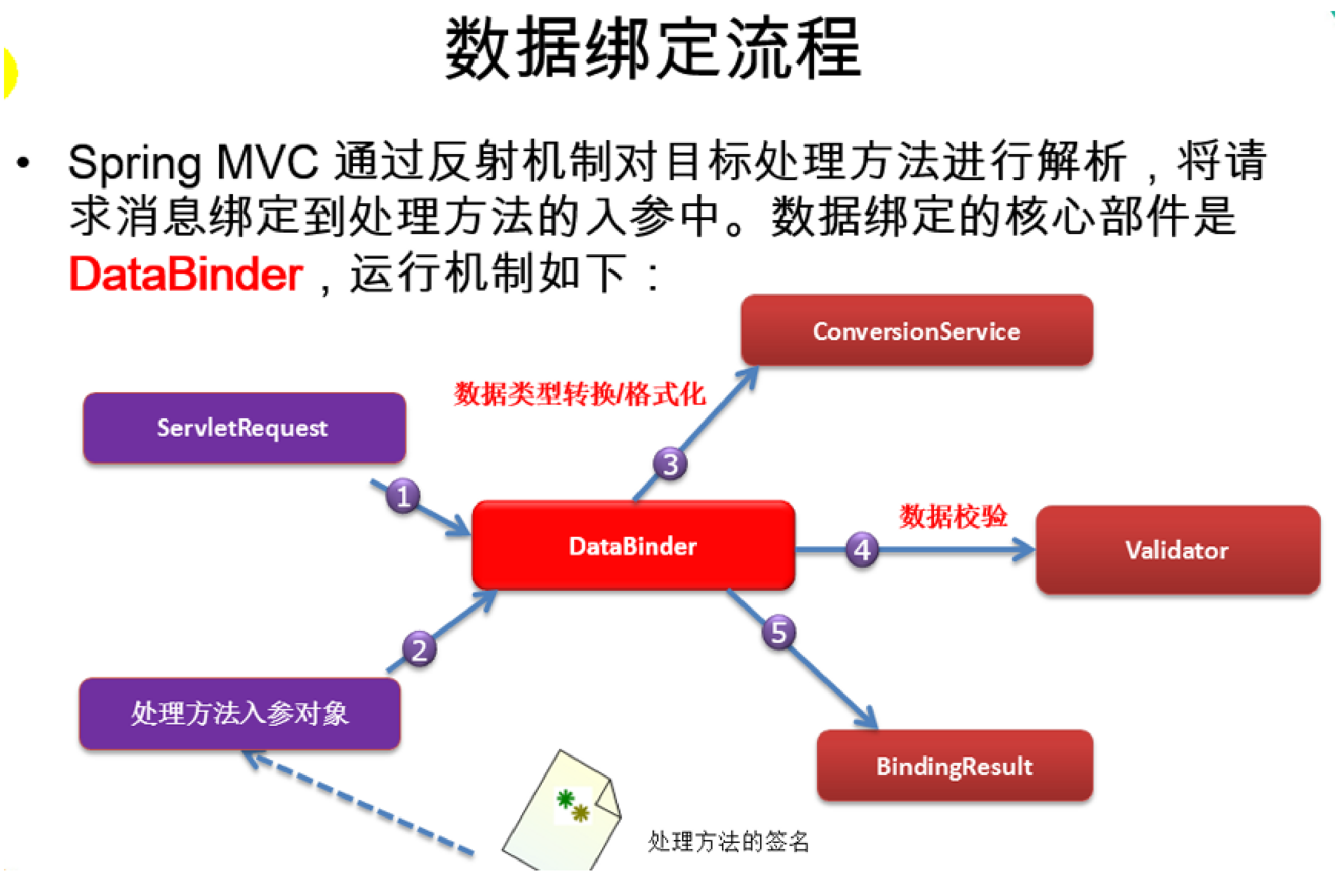
```
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat
java.time.LocalDateTime:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.lang.String -> java.time.LocalDateTime:
org.springframework.format.datetime.standard.TemporalAccessorParser@6bf61650
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat java.time.LocalTime:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.lang.String -> java.time.LocalTime:
org.springframework.format.datetime.standard.TemporalAccessorParser@2837beeb
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat
java.time.OffsetDateTime:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.lang.String -> java.time.OffsetDateTime:
org.springframework.format.datetime.standard.TemporalAccessorParser@60f3f7
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat
java.time.OffsetTime:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.lang.String -> java.time.OffsetTime:
org.springframework.format.datetime.standard.TemporalAccessorParser@3cb8bdb7
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat
java.time.ZonedDateTime:
org.springframework.format.datetime.standard.Jsr310DateTimeFormatAnnotationFormatterFactory@39274b
77,java.lang.String -> java.time.ZonedDateTime:
org.springframework.format.datetime.standard.TemporalAccessorParser@14a1511f
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat java.util.Calendar:
org.springframework.format.datetime.DateTimeFormatAnnotationFormatterFactory@32abc654
    java.lang.String -> @org.springframework.format.annotation.DateTimeFormat java.util.Date:
org.springframework.format.datetime.DateTimeFormatAnnotationFormatterFactory@32abc654
    java.lang.String -> @org.springframework.format.annotation.NumberFormat java.lang.Double:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.String -> @org.springframework.format.annotation.NumberFormat java.lang.Float:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.String -> @org.springframework.format.annotation.NumberFormat java.lang.Integer:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.String -> @org.springframework.format.annotation.NumberFormat java.lang.Short:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.String -> @org.springframework.format.annotation.NumberFormat java.math.BigDecimal:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.String -> @org.springframework.format.annotation.NumberFormat java.math.BigInteger:
org.springframework.format.number.NumberFormatAnnotationFormatterFactory@140bb45d
    java.lang.String -> java.lang.Boolean :
org.springframework.core.convert.support.StringToBooleanConverter@22f562e2
    java.lang.String -> java.lang.Character :
org.springframework.core.convert.support.StringToCharacterConverter@5f2594f5
    java.lang.String -> java.lang.Enum :
org.springframework.core.convert.support.StringToEnumConverterFactory@1347a7be
    java.lang.String -> java.lang.Number :
org.springframework.core.convert.support.StringToNumberConverterFactory@28a5e291
    java.lang.String -> java.time.Instant:
org.springframework.format.datetime.standard.InstantFormatter@1eb74d34
    java.lang.String -> java.util.Locale :
org.springframework.core.convert.support.StringToLocaleConverter@6def03d3
    java.lang.String -> java.util.Properties :
org.springframework.core.convert.support.StringToPropertiesConverter@569d2e80
    java.lang.String -> java.util.UUID :
org.springframework.core.convert.support.StringToUUIDConverter@343b18b
```

```
    java.time.Instant -> java.lang.String :
org.springframework.format.datetime.standard.InstantFormatter@1eb74d34
    java.time.ZoneId -> java.util.TimeZone :
org.springframework.core.convert.support.ZoneIdToTimeZoneConverter@63b3b722
    java.util.Calendar -> java.lang.Long :
org.springframework.format.datetime.DateFormatterRegistrar$CalendarToLongConverter@1f07f950
    java.util.Calendar -> java.util.Date :
org.springframework.format.datetime.DateFormatterRegistrar$CalendarToDateConverter@3bfdcdf6
    java.util.Date -> java.lang.Long :
org.springframework.format.datetime.DateFormatterRegistrar$DateToLongConverter@19cfea5e
    java...
```

以java.lang.String -> java.lang.Boolean String转为boolean为例 其他的可以自行跟踪代码

```java
final class StringToBooleanConverter implements Converter<String, Boolean> {

    //true集合
    private static final Set<String> trueValues = new HashSet<String>(4);

    //false集合
    private static final Set<String> falseValues = new HashSet<String>(4);
    //由这个初始化方法可知对于true这一项而言 页面传的值为'true','on','yes','1'都是可以的
    static {
        trueValues.add("true");
        trueValues.add("on");
        trueValues.add("yes");
        trueValues.add("1");

        falseValues.add("false");
        falseValues.add("off");
        falseValues.add("no");
        falseValues.add("0");
    }

    //转化器  真正工作的方法
    @Override
    public Boolean convert(String source) {
        String value = source.trim();
        if ("".equals(value)) {
            return null;
        }
        value = value.toLowerCase();
        if (trueValues.contains(value)) {
            return Boolean.TRUE;
        }
        else if (falseValues.contains(value)) {
            return Boolean.FALSE;
        }
        else {
            throw new IllegalArgumentException("Invalid boolean value '" + source + "'");
        }
    }

}
```

数据绑定的整体流程图



数据绑定流程

- Spring MVC 通过反射机制对目标处理方法进行解析，将请求消息绑定到处理方法的入参中。数据绑定的核心部件是 DataBinder，运行机制如下：

自定义数据转化器:

自定义类型装换器需要写的大体步骤: 1.实现Converter接口，写一个自定义的类型转换器； 2.配置出ConversionService；

```xml
<!-- 告诉SpringMVC别用默认的ConversionService,
            而用我自定义的ConversionService、可能有我们自定义的Converter;  -->
<!--ConversionServiceFactoryBean 这个有个确定就是只有转化器 没有格式化器 后续会介绍怎么改 -->
<bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
    <!--converters转换器中添加我们自定义的类型转换器  -->
    <property name="converters">
        <set>
            <bean class="com.mgw.component.MyStringToEmployeeConverter"></bean>
        </set>
    </property>
</bean>
```

```java
//上述配置说到了ConversionServiceFactoryBean  那就顺便说说吧
public class ConversionServiceFactoryBean implements FactoryBean<ConversionService>,
InitializingBean {

    private Set<?> converters;

    private GenericConversionService conversionService;

    //设置一些converters
    public void setConverters(Set<?> converters) {
        this.converters = converters;
    }

    //注意这个方法  这个方法是在整个bean刚创建后 初始化时调用的 在spring源码时有讲
    @Override
    public void afterPropertiesSet() {
        //设置conversionService
        this.conversionService = createConversionService();
        //并将converters 和 conversionService注册进工厂
        ConversionServiceFactory.registerConverters(this.converters, this.conversionService);
    }


    protected GenericConversionService createConversionService() {
        return new DefaultConversionService();
    }


    // implementing FactoryBean

    //拿到一个conversionService对象
    @Override
    public ConversionService getObject() {
        return this.conversionService;
    }

    @Override
    public Class<? extends ConversionService> getObjectType() {
        return GenericConversionService.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }

}
```

3.让SpringMVC用我们的ConversionService

```
<!-- conversion-service="conversionService"：使用我们自己配置的类型转换组件 -->
<mvc:annotation-driven conversion-service="conversionService"></mvc:annotation-driven>
```

Converter与ConversionService的两点说明: 1.ConversionService:是一个接口； 它里面有Converter（转换器）进行工作: (上面已经进行分析过了) 2.Converter是ConversionService中的组件； 1、你的Converter得放进ConversionService 中； 2、将WebDataBinder中的ConversionService设置成我们这个加了自定义类型转换器的ConversionService；

说明：将页面上传递的数据"empAdmin-admin@qq.com-1-101" 转为一个Employee对象

```
public class Department {

    private Integer id;
    private String departmentName;
    //TODO ...
    //省略set get方法
}

public class Employee {

    private Integer id;

    @NotEmpty(message="不能为空")
    @Length(min=5,max=17,message="我错了")
    private String lastName;


    @Email
    private String email;
    //1 male, 0 female
    private Integer gender;


    //规定页面提交的日期格式
    //@Past：必须是一个过去的时间
    //@Future ：必须是一个未来的时间
    //页面提交的数据格式如果不正确，就是400；
    @DateTimeFormat(pattern="yyyy-MM-dd")
    @Past
    private Date birth = new Date();

    //假设页面，为了显示方便提交的工资是  ￥10,000.98
    @NumberFormat(pattern="#,###.##")
    private Double salary;

    @JsonIgnore
    private Department department;

    //TODO ...
    //省略set get方法
}
//自定义转化器 实现Converter接口
public class MyStringToEmployeeConverter implements Converter<String, Employee> {

    @Autowired
    DepartmentDao departmentDao;

    /**
     * 自定义的转换规则
     */
    @Override
    public Employee convert(String source) {
        // empAdmin-admin@qq.com-1-101
        System.out.println("页面提交的将要转换的字符串" + source);
        Employee employee = new Employee();
        if (source.contains("-")) {
            String[] split = source.split("-");
            employee.setLastName(split[0]);
            employee.setEmail(split[1]);
            employee.setGender(Integer.parseInt(split[2]));
            employee.setDepartment(departmentDao.getDepartment(Integer.parseInt(split[3])));
        }
        return employee;
    }

}
```

spring-mvc.xml的配置

```
<!-- conversion-service="conversionService"：使用我们自己配置的类型转换组件 -->
<mvc:annotation-driven conversion-service="conversionService"></mvc:annotation-driven>

<!-- 告诉SpringMVC别用默认的ConversionService，
   而用我自定义的ConversionService、可能有我们自定义的Converter；  -->
<bean id="conversionService"
class="org.springframework.format.support.ConversionServiceFactoryBean">
    <!--converters转换器中添加我们自定义的类型转换器  -->
    <property name="converters">
        <set>
            <bean class="com.atguigu.component.MyStringToEmployeeConverter"></bean>
        </set>
    </property>
</bean>
```

总结三步： 1.实现Converter接口，做一个自定义类型的转换器 2.将这个Converter配置在ConversionService中 3.告诉SpringMVC使用这个ConversionService

前面已经说过了ConversionServiceFactoryBean这个bean只有转化器没有有格式化器 这里对其进行改造 改变使用的bean工厂即可

```
<!-- conversion-service="conversionService"：使用我们自己配置的类型转换组件 -->
<mvc:annotation-driven conversion-service="conversionService"></mvc:annotation-driven>

<!-- 告诉SpringMVC别用默认的ConversionService，
   而用我自定义的ConversionService、可能有我们自定义的Converter；  -->
<!-- 以后写自定义类型转换器的时候，就使用FormattingConversionServiceFactoryBean来注册；
 既具有类型转换还有格式化功能 -->
<bean id="conversionService"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <!--converters转换器中添加我们自定义的类型转换器  -->
    <property name="converters">
        <set>
            <bean class="com.atguigu.component.MyStringToEmployeeConverter"></bean>
        </set>
    </property>
</bean>
```

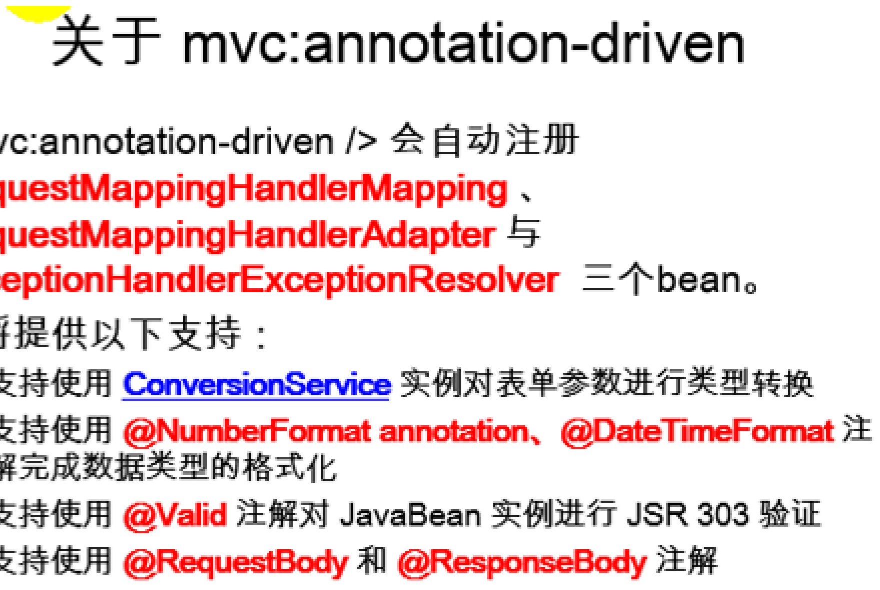
```
//同样我们来说说FormattingConversionServiceFactoryBean这个这个类
public class FormattingConversionServiceFactoryBean
        implements FactoryBean<FormattingConversionService>, EmbeddedValueResolverAware,
InitializingBean {

    //转化器集合
    private Set<?> converters;
```

```java
//格式化器集合
private Set<?> formatters;

private Set<FormatterRegistrar> formatterRegistrars;

private boolean registerDefaultFormatters = true;

private StringValueResolver embeddedValueResolver;

private FormattingConversionService conversionService;


/**
 * Configure the set of custom converter objects that should be added.
 * @param converters instances of any of the following:
 * {@link org.springframework.core.convert.converter.Converter},
 * {@link org.springframework.core.convert.converter.ConverterFactory},
 * {@link org.springframework.core.convert.converter.GenericConverter}
 */
public void setConverters(Set<?> converters) {
    this.converters = converters;
}

/**
 * Configure the set of custom formatter objects that should be added.
 * @param formatters instances of {@link Formatter} or {@link AnnotationFormatterFactory}
 */
public void setFormatters(Set<?> formatters) {
    this.formatters = formatters;
}

/**
 * <p>Configure the set of FormatterRegistrars to invoke to register
 * Converters and Formatters in addition to those added declaratively
 * via {@link #setConverters(Set)} and {@link #setFormatters(Set)}.
 * <p>FormatterRegistrars are useful when registering multiple related
 * converters and formatters for a formatting category, such as Date
 * formatting. All types related needed to support the formatting
 * category can be registered from one place.
 * <p>FormatterRegistrars can also be used to register Formatters
 * indexed under a specific field type different from its own &lt;T&gt;,
 * or when registering a Formatter from a Printer/Parser pair.
 * @see FormatterRegistry#addFormatterForFieldType(Class, Formatter)
 * @see FormatterRegistry#addFormatterForFieldType(Class, Printer, Parser)
 */
public void setFormatterRegistrars(Set<FormatterRegistrar> formatterRegistrars) {
    this.formatterRegistrars = formatterRegistrars;
}

/**
 * Indicate whether default formatters should be registered or not.
 * <p>By default, built-in formatters are registered. This flag can be used
 * to turn that off and rely on explicitly registered formatters only.
 * @see #setFormatters(Set)
 * @see #setFormatterRegistrars(Set)
 */
```

```
    public void setRegisterDefaultFormatters(boolean registerDefaultFormatters) {
        this.registerDefaultFormatters = registerDefaultFormatters;
    }

    @Override
    public void setEmbeddedValueResolver(StringValueResolver embeddedValueResolver) {
        this.embeddedValueResolver = embeddedValueResolver;
    }

    //注意这个方法  这个方法是在整个bean刚创建后 初始化时调用的 在spring源码时有讲
    @Override
    public void afterPropertiesSet() {
        this.conversionService = new
DefaultFormattingConversionService(this.embeddedValueResolver, this.registerDefaultFormatters);
        //注册转化器
        ConversionServiceFactory.registerConverters(this.converters, this.conversionService);
        //注册格式化器
        registerFormatters();
    }

    private void registerFormatters() {
        if (this.formatters != null) {
            for (Object formatter : this.formatters) {
                if (formatter instanceof Formatter<?>) {
                    this.conversionService.addFormatter((Formatter<?>) formatter);
                }
                else if (formatter instanceof AnnotationFormatterFactory<?>) {

this.conversionService.addFormatterForFieldAnnotation((AnnotationFormatterFactory<?>) formatter);
                }
                else {
                    throw new IllegalArgumentException(
                            "Custom formatters must be implementations of Formatter or
AnnotationFormatterFactory");
                }
            }
        }
        if (this.formatterRegistrars != null) {
            for (FormatterRegistrar registrar : this.formatterRegistrars) {
                registrar.registerFormatters(this.conversionService);
            }
        }
    }


    @Override
    public FormattingConversionService getObject() {
        return this.conversionService;
    }

    @Override
    public Class<? extends FormattingConversionService> getObjectType() {
        return FormattingConversionService.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }

}
```

```
SpringMVC解析<mvc:annotation-driven ></mvc:annotation-driven>表签到底做了哪些事情?
```

这个标签所做的事情如下图:



关于 mvc:annotation-driven

- <mvc:annotation-driven /> 会自动注册 RequestMappingHandlerMapping 、RequestMappingHandlerAdapter 与 ExceptionHandlerExceptionResolver 三个bean。
- 还将提供以下支持：
  - 支持使用 ConversionService 实例对表单参数进行类型转换
  - 支持使用 @NumberFormat annotation、@DateTimeFormat 注解完成数据类型的格式化
  - 支持使用 @Valid 注解对 JavaBean 实例进行 JSR 303 验证
  - 支持使用 @RequestBody 和 @ResponseBody 注解

这个标签有点像springMvc中的一款外挂，其功能很全很强大

顺便说一下 我没有加这个<mvc:annotation-driven >标签前我的处理器映射使用的是DefaultAnnotationHandlerMapping，处理器适配器用的是AnnotationMethodHandlerAdapter 查看类时发现标注的是过时 但是加了<mvc:annotation-driven >标签后我的处理器映射使用的是RequestMappingHandlerMapping,处理器适配器使用的是RequestMappingHandlerAdapter 。由此可见其实后面加了<mvc:annotation-driven >标签后的的这两个其实就是用于取代前者的 springMvc更加推荐你使用这个标签。

说下这个标签为啥会添加这么多的东西?

```
//在BeanDefinitionParser(所有的BeanDefinition解析器的总接口)这个接口下有
AnnotationDrivenBeanDefinitionParser这个类来解析<mvc:annotation-driven >这个标签
//先提取他里面重要的一些东西进行部分分析  其实这个标签太强大  我也没弄的很清晰  后面继续搞  继续更新
public static final String HANDLER_MAPPING_BEAN_NAME =
RequestMappingHandlerMapping.class.getName();

public static final String HANDLER_ADAPTER_BEAN_NAME =
RequestMappingHandlerAdapter.class.getName();

public static final String CONTENT_NEGOTIATION_MANAGER_BEAN_NAME = "mvcContentNegotiationManager";

//数据校验相关类是否存在
private static final boolean javaxValidationPresent =
        ClassUtils.isPresent("javax.validation.Validator",
                AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

private static boolean romePresent =
        ClassUtils.isPresent("com.rometools.rome.feed.WireFeed",
                AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

private static final boolean jaxb2Present =
        ClassUtils.isPresent("javax.xml.bind.Binder",
                AnnotationDrivenBeanDefinitionParser.class.getClassLoader());
//转json相关类是否存在
private static final boolean jackson2Present =
        ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper",
                AnnotationDrivenBeanDefinitionParser.class.getClassLoader()) &&
        ClassUtils.isPresent("com.fasterxml.jackson.core.JsonGenerator",
                AnnotationDrivenBeanDefinitionParser.class.getClassLoader());
//json互转xml相关类是否存在
private static final boolean jackson2XmlPresent =
        ClassUtils.isPresent("com.fasterxml.jackson.dataformat.xml.XmlMapper",
                AnnotationDrivenBeanDefinitionParser.class.getClassLoader());
//转gson相关类是否存在
private static final boolean gsonPresent =
        ClassUtils.isPresent("com.google.gson.Gson",
                AnnotationDrivenBeanDefinitionParser.class.getClassLoader());

public BeanDefinition parse(Element element, ParserContext parserContext) {
    Object source = parserContext.extractSource(element);
    XmlReaderContext readerContext = parserContext.getReaderContext();

    CompositeComponentDefinition compDefinition = new
CompositeComponentDefinition(element.getTagName(), source);
    parserContext.pushContainingComponent(compDefinition);

    RuntimeBeanReference contentNegotiationManager = getContentNegotiationManager(element, source,
parserContext);

    RootBeanDefinition handlerMappingDef = new
RootBeanDefinition(RequestMappingHandlerMapping.class);
    handlerMappingDef.setSource(source);
    handlerMappingDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    handlerMappingDef.getPropertyValues().add("order", 0);
    handlerMappingDef.getPropertyValues().add("contentNegotiationManager",
contentNegotiationManager);

    if (element.hasAttribute("enable-matrix-variables")) {
        Boolean enableMatrixVariables = Boolean.valueOf(element.getAttribute("enable-matrix-
variables"));
        handlerMappingDef.getPropertyValues().add("removeSemicolonContent",
!enableMatrixVariables);
    }
    else if (element.hasAttribute("enableMatrixVariables")) {
        Boolean enableMatrixVariables =
Boolean.valueOf(element.getAttribute("enableMatrixVariables"));
        handlerMappingDef.getPropertyValues().add("removeSemicolonContent",
!enableMatrixVariables);
    }

    configurePathMatchingProperties(handlerMappingDef, element, parserContext);
```

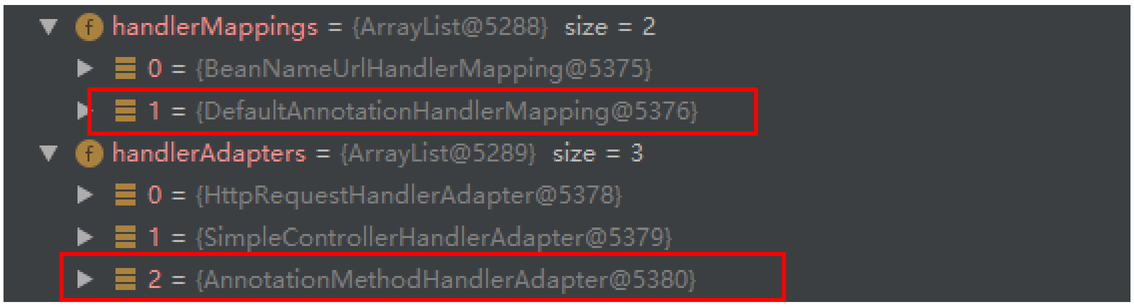
```java
    readerContext.getRegistry().registerBeanDefinition(HANDLER_MAPPING_BEAN_NAME ,
handlerMappingDef);

    RuntimeBeanReference corsRef = MvcNamespaceUtils.registerCorsConfigurations(null,
parserContext, source);
    handlerMappingDef.getPropertyValues().add("corsConfigurations", corsRef);

    RuntimeBeanReference conversionService = getConversionService(element, source, parserContext);
    RuntimeBeanReference validator = getValidator(element, source, parserContext);
    RuntimeBeanReference messageCodesResolver = getMessageCodesResolver(element);

    RootBeanDefinition bindingDef = new
RootBeanDefinition(ConfigurableWebBindingInitializer.class);
    bindingDef.setSource(source);
    bindingDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    bindingDef.getPropertyValues().add("conversionService", conversionService);
    bindingDef.getPropertyValues().add("validator", validator);
    bindingDef.getPropertyValues().add("messageCodesResolver", messageCodesResolver);

    ManagedList<?> messageConverters = getMessageConverters(element, source, parserContext);
    ManagedList<?> argumentResolvers = getArgumentResolvers(element, parserContext);
    ManagedList<?> returnValueHandlers = getReturnValueHandlers(element, parserContext);
    String asyncTimeout = getAsyncTimeout(element);
    RuntimeBeanReference asyncExecutor = getAsyncExecutor(element);
    ManagedList<?> callableInterceptors = getCallableInterceptors(element, source, parserContext);
    ManagedList<?> deferredResultInterceptors = getDeferredResultInterceptors(element, source,
parserContext);

    RootBeanDefinition handlerAdapterDef = new
RootBeanDefinition(RequestMappingHandlerAdapter.class);
    handlerAdapterDef.setSource(source);
    handlerAdapterDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    handlerAdapterDef.getPropertyValues().add("contentNegotiationManager",
contentNegotiationManager);
    handlerAdapterDef.getPropertyValues().add("webBindingInitializer", bindingDef);
    handlerAdapterDef.getPropertyValues().add("messageConverters", messageConverters);
    addRequestBodyAdvice(handlerAdapterDef);
    addResponseBodyAdvice(handlerAdapterDef);

    if (element.hasAttribute("ignore-default-model-on-redirect")) {
        Boolean ignoreDefaultModel = Boolean.valueOf(element.getAttribute("ignore-default-model-
on-redirect"));
        handlerAdapterDef.getPropertyValues().add("ignoreDefaultModelOnRedirect",
ignoreDefaultModel);
    }
    else if (element.hasAttribute("ignoreDefaultModelOnRedirect")) {
        // "ignoreDefaultModelOnRedirect" spelling is deprecated
        Boolean ignoreDefaultModel =
Boolean.valueOf(element.getAttribute("ignoreDefaultModelOnRedirect"));
        handlerAdapterDef.getPropertyValues().add("ignoreDefaultModelOnRedirect",
ignoreDefaultModel);
    }

    if (argumentResolvers != null) {
        handlerAdapterDef.getPropertyValues().add("customArgumentResolvers", argumentResolvers);
    }
    if (returnValueHandlers != null) {
        handlerAdapterDef.getPropertyValues().add("customReturnValueHandlers",
returnValueHandlers);
    }
    if (asyncTimeout != null) {
        handlerAdapterDef.getPropertyValues().add("asyncRequestTimeout", asyncTimeout);
    }
    if (asyncExecutor != null) {
        handlerAdapterDef.getPropertyValues().add("taskExecutor", asyncExecutor);
    }

    handlerAdapterDef.getPropertyValues().add("callableInterceptors", callableInterceptors);
    handlerAdapterDef.getPropertyValues().add("deferredResultInterceptors",
deferredResultInterceptors);
    readerContext.getRegistry().registerBeanDefinition(HANDLER_ADAPTER_BEAN_NAME ,
handlerAdapterDef);

    RootBeanDefinition uriContributorDef =
            new RootBeanDefinition(CompositeUriComponentsContributorFactoryBean.class);
    uriContributorDef.setSource(source);
    uriContributorDef.getPropertyValues().addPropertyValue("handlerAdapter", handlerAdapterDef);
    uriContributorDef.getPropertyValues().addPropertyValue("conversionService",
conversionService);
    String uriContributorName = MvcUriComponentsBuilder.MVC_URI_COMPONENTS_CONTRIBUTOR_BEAN_NAME;
    readerContext.getRegistry().registerBeanDefinition(uriContributorName, uriContributorDef);

    RootBeanDefinition csInterceptorDef = new
RootBeanDefinition(ConversionServiceExposingInterceptor.class);
    csInterceptorDef.setSource(source);
    csInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(0, conversionService);
    RootBeanDefinition mappedInterceptorDef = new RootBeanDefinition(MappedInterceptor.class);
    mappedInterceptorDef.setSource(source);
    mappedInterceptorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    mappedInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(0, (Object) null);
    mappedInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(1,
csInterceptorDef);
    String mappedInterceptorName = readerContext.registerWithGeneratedName(mappedInterceptorDef);
    //添加ExceptionHandlerExceptionResolver异常解析器用来替代AnnotationMethodHandlerExceptionResolver
    RootBeanDefinition methodExceptionResolver = new
RootBeanDefinition(ExceptionHandlerExceptionResolver.class);
    methodExceptionResolver.setSource(source);
    methodExceptionResolver.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    methodExceptionResolver.getPropertyValues().add("contentNegotiationManager",
contentNegotiationManager);
    methodExceptionResolver.getPropertyValues().add("messageConverters", messageConverters);
    methodExceptionResolver.getPropertyValues().add("order", 0);
    addResponseBodyAdvice(methodExceptionResolver);
    if (argumentResolvers != null) {
        methodExceptionResolver.getPropertyValues().add("customArgumentResolvers",
argumentResolvers);
    }
    if (returnValueHandlers != null) {
        methodExceptionResolver.getPropertyValues().add("customReturnValueHandlers",
returnValueHandlers);
    }
    String methodExResolverName =
readerContext.registerWithGeneratedName(methodExceptionResolver);
```

```
    RootBeanDefinition statusExceptionResolver = new
RootBeanDefinition(ResponseStatusExceptionResolver.class);
    statusExceptionResolver.setSource(source);
    statusExceptionResolver.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    statusExceptionResolver.getPropertyValues().add("order", 1);
    String statusExResolverName =
readerContext.registerWithGeneratedName(statusExceptionResolver);

    RootBeanDefinition defaultExceptionResolver = new
RootBeanDefinition(DefaultHandlerExceptionResolver.class);
    defaultExceptionResolver.setSource(source);
    defaultExceptionResolver.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    defaultExceptionResolver.getPropertyValues().add("order", 2);
    String defaultExResolverName =
readerContext.registerWithGeneratedName(defaultExceptionResolver);

    //添加RequestMappingHandlerMapping
    parserContext.registerComponent(new BeanComponentDefinition(handlerMappingDef,
HANDLER_MAPPING_BEAN_NAME));
    //添加RequestMappingHandlerAdapter
    parserContext.registerComponent(new BeanComponentDefinition(handlerAdapterDef,
HANDLER_ADAPTER_BEAN_NAME));
    parserContext.registerComponent(new BeanComponentDefinition(uriContributorDef,
uriContributorName));
    parserContext.registerComponent(new BeanComponentDefinition(mappedInterceptorDef,
mappedInterceptorName));
    parserContext.registerComponent(new BeanComponentDefinition(methodExceptionResolver,
methodExResolverName));
    parserContext.registerComponent(new BeanComponentDefinition(statusExceptionResolver,
statusExResolverName));
    parserContext.registerComponent(new BeanComponentDefinition(defaultExceptionResolver,
defaultExResolverName));

    // Ensure BeanNameUrlHandlerMapping (SPR-8289) and default HandlerAdapters are not "turned
off"
    MvcNamespaceUtils.registerDefaultComponents(parserContext, source);

    parserContext.popAndRegisterContainingComponent();

    return null;
}
```

在本文的开头讲配置拦截/和/*时讲到"mvc:default-servlet-handler/" " mvc:annotation-driven/"同时配置时才能既可以处理静态资源又可以处理动态资源，下面做个详细的说明

"mvc:default-servlet-handler/ mvc:annotation-driven/"配置情况不同的现象:

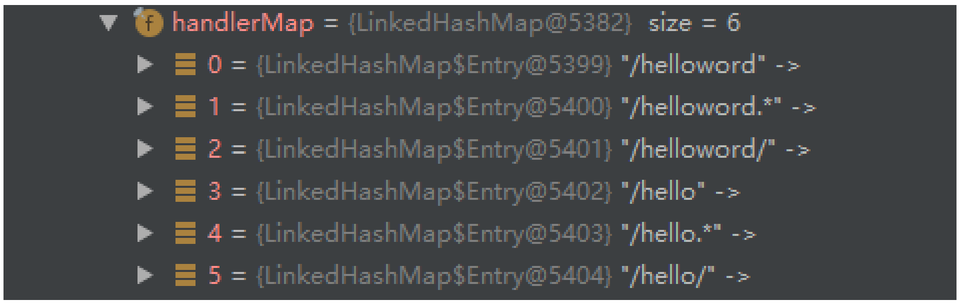
1.都没配时 动态资源@RequestMapping映射的资源能访问，静态资源（.html,.js,.img）不能访问


```
▼ handlerMappings = {ArrayList@5288}  size = 2
    ▶ 0 = {BeanNameUrlHandlerMapping@5375}
    ▶ 1 = {DefaultAnnotationHandlerMapping@5376}
▼ handlerAdapters = {ArrayList@5289}  size = 3
    ▶ 0 = {HttpRequestHandlerAdapter@5378}
    ▶ 1 = {SimpleControllerHandlerAdapter@5379}
    ▶ 2 = {AnnotationMethodHandlerAdapter@5380}
```

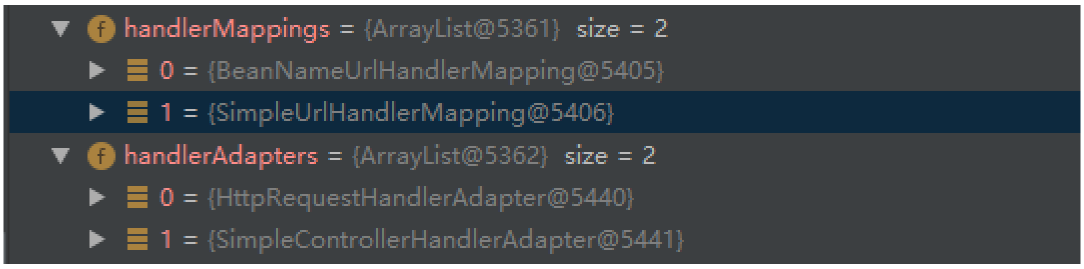
此时这是我们使用的HandlerMapping和HandlerAdapter

此时的HandlerMapping中的handlerMap中保存了每一个了每一个资源的映射信息所以动态资源就可以

静态不能访问,就是handlerMap中没有保存静态资源映射的请求.


```
▼ handlerMap = {LinkedHashMap@5382}  size = 6
    ▶ 0 = {LinkedHashMap$Entry@5399} "/helloword" ->
    ▶ 1 = {LinkedHashMap$Entry@5400} "/helloword.*" ->
    ▶ 2 = {LinkedHashMap$Entry@5401} "/helloword/" ->
    ▶ 3 = {LinkedHashMap$Entry@5402} "/hello" ->
    ▶ 4 = {LinkedHashMap$Entry@5403} "/hello.*" ->
    ▶ 5 = {LinkedHashMap$Entry@5404} "/hello/" ->
```

2.加上"mvc:default-servlet-handler/"，不加"mvc:annotation-driven/ "时 ,静态资源ok，动态资源完蛋


```
▼ handlerMappings = {ArrayList@5361}  size = 2
    ▶ 0 = {BeanNameUrlHandlerMapping@5405}
    ▶ 1 = {SimpleUrlHandlerMapping@5406}
▼ handlerAdapters = {ArrayList@5362}  size = 2
    ▶ 0 = {HttpRequestHandlerAdapter@5440}
    ▶ 1 = {SimpleControllerHandlerAdapter@5441}
```

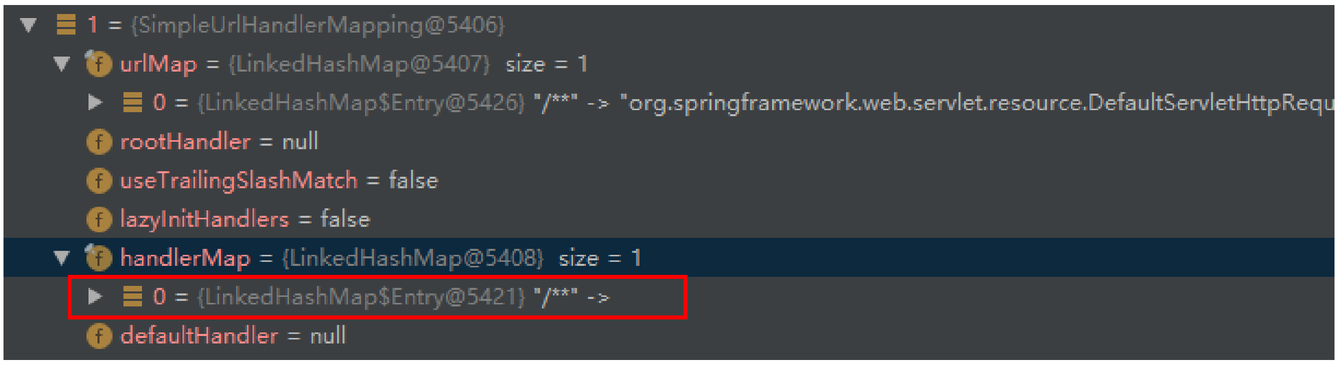
此时的HandlerMapping中的handlerMap都变了并且只有一个映射/**

且"/**" -> "org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler#0" 意思就是所有的请求都交给tomcat去处理

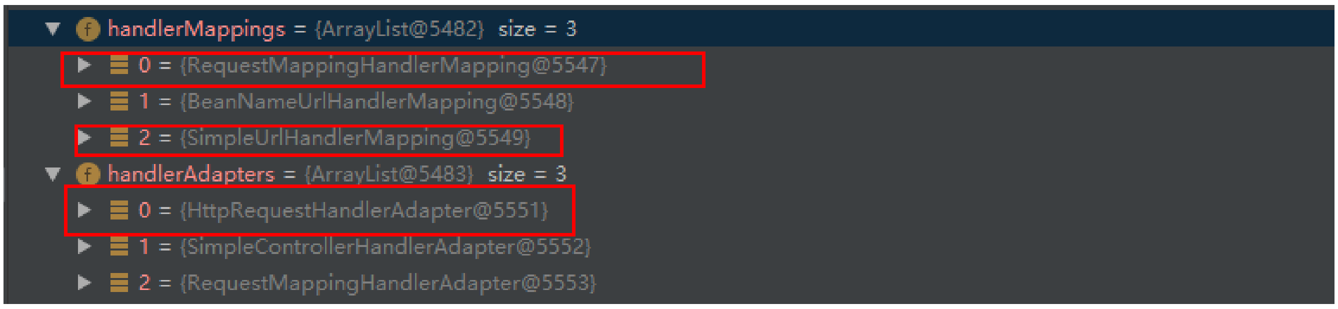
但是tomcat只处理静态资源啊 不能处理动态资源 所以自然无法处理动态资源

动态不能访问的原因：DefaultAnnotationHandlerMapping没有了；用SimpleUrlHandlerMapping替换了，他的作用就是将所有请求直接交给tomcat；

静态能访问的原因：SimpleUrlHandlerMapping把所有请求都映射给tomcat；

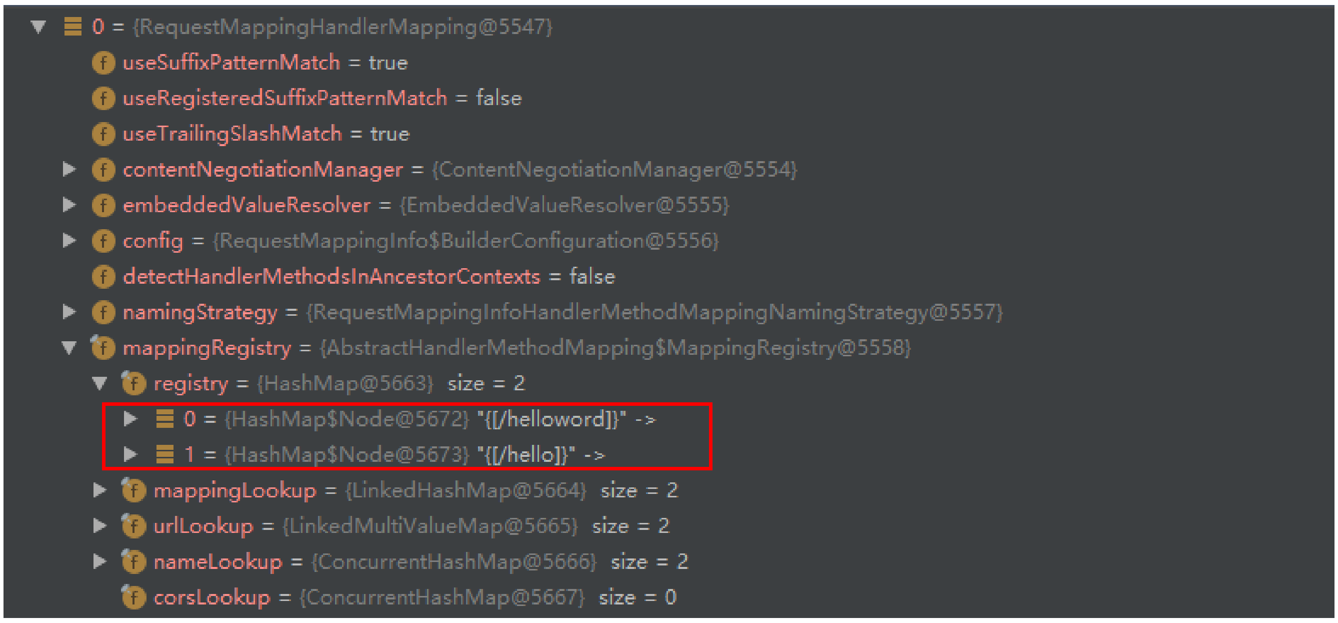

```
▼ 1 = {SimpleUrlHandlerMapping@5406}
    ▼ urlMap = {LinkedHashMap@5407}  size = 1
        ▶ 0 = {LinkedHashMap$Entry@5426} "/**" -> "org.springframework.web.servlet.resource.DefaultServletHttpRequ
      rootHandler = null
      useTrailingSlashMatch = false
      lazyInitHandlers = false
    ▼ handlerMap = {LinkedHashMap@5408}  size = 1
        ▶ 0 = {LinkedHashMap$Entry@5421} "/**" ->
      defaultHandler = null
```

3.加上“mvc:default-servlet-handler/”，加上"mvc:annotation-driven/" 时,动态资源ok，静态资源也ok

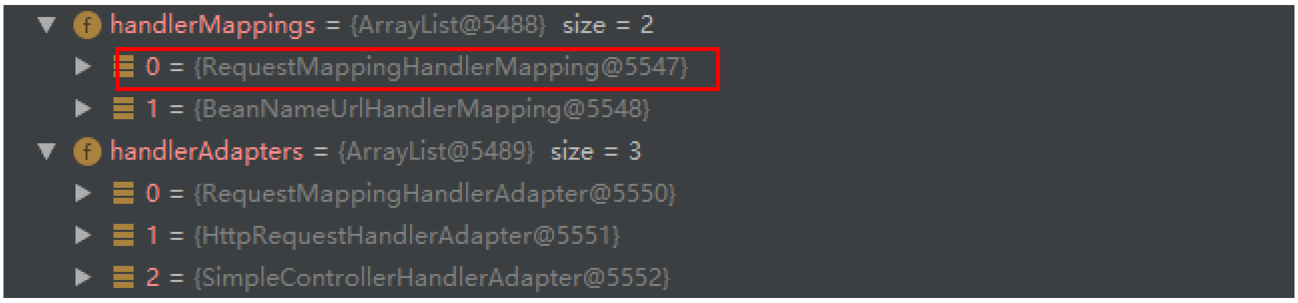

```
▼ handlerMappings = {ArrayList@5482}  size = 3
    ▶ 0 = {RequestMappingHandlerMapping@5547}
    ▶ 1 = {BeanNameUrlHandlerMapping@5548}
    ▶ 2 = {SimpleUrlHandlerMapping@5549}
▼ handlerAdapters = {ArrayList@5483}  size = 3
    ▶ 0 = {HttpRequestHandlerAdapter@5551}
    ▶ 1 = {SimpleControllerHandlerAdapter@5552}
    ▶ 2 = {RequestMappingHandlerAdapter@5553}
```

此时多了一个RequestMappingHandlerMapping这个处理器映射器还有原来的SimpleUrlHandlerMapping(处理静态资源)

这个RequestMappingHandlerMapping里就保存了各个动态资源请求的处理方法映射等 所以可以处理动态资源


```
▼ 0 = {RequestMappingHandlerMapping@5547}
      useSuffixPatternMatch = true
      useRegisteredSuffixPatternMatch = false
      useTrailingSlashMatch = true
  ▶   contentNegotiationManager = {ContentNegotiationManager@5554}
  ▶   embeddedValueResolver = {EmbeddedValueResolver@5555}
  ▶   config = {RequestMappingInfo$BuilderConfiguration@5556}
      detectHandlerMethodsInAncestorContexts = false
  ▶   namingStrategy = {RequestMappingInfoHandlerMethodMappingNamingStrategy@5557}
  ▼   mappingRegistry = {AbstractHandlerMethodMapping$MappingRegistry@5558}
    ▼   registry = {HashMap@5663}  size = 2
      ▶   0 = {HashMap$Node@5672} "{[/helloword]}" ->
      ▶   1 = {HashMap$Node@5673} "{[/hello]}" ->
    ▶   mappingLookup = {LinkedHashMap@5664}  size = 2
    ▶   urlLookup = {LinkedMultiValueMap@5665}  size = 2
    ▶   nameLookup = {ConcurrentHashMap@5666}  size = 2
        corsLookup = {ConcurrentHashMap@5667}  size = 0
```

4.只加"mvc:annotation-driven/" 时,静态资源完蛋,动态资源ok

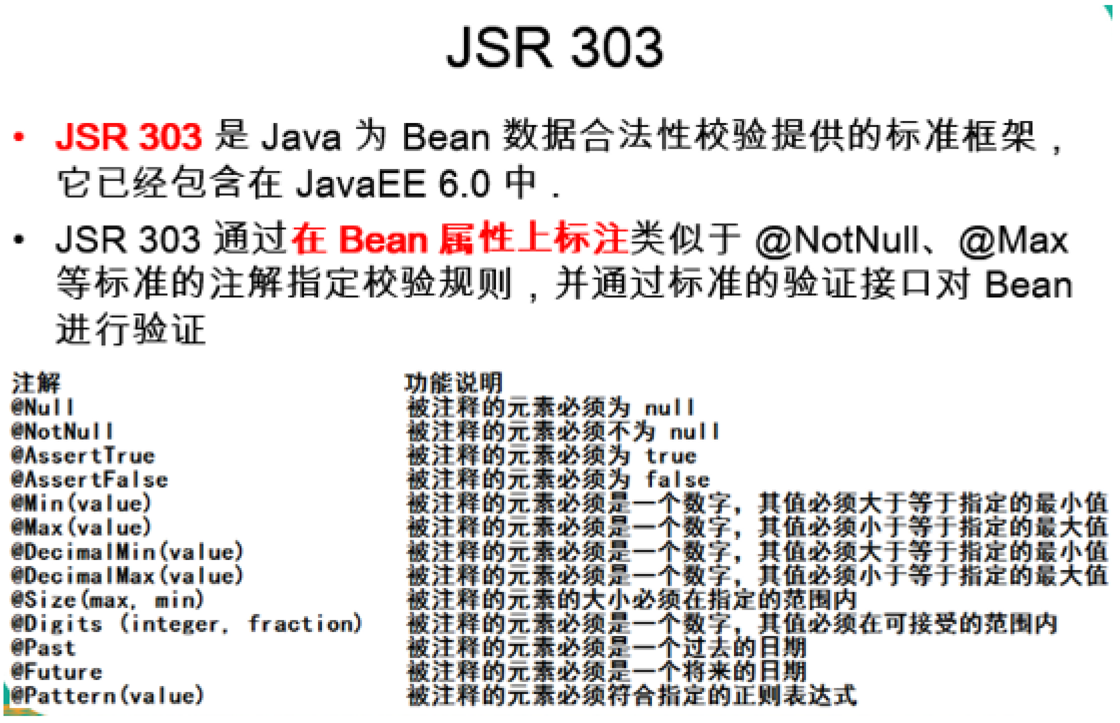

```
▼ handlerMappings = {ArrayList@5488}  size = 2
    ▶ 0 = {RequestMappingHandlerMapping@5547}
    ▶ 1 = {BeanNameUrlHandlerMapping@5548}
▼ handlerAdapters = {ArrayList@5489}  size = 3
    ▶ 0 = {RequestMappingHandlerAdapter@5550}
    ▶ 1 = {HttpRequestHandlerAdapter@5551}
    ▶ 2 = {SimpleControllerHandlerAdapter@5552}
```

此时没有SimpleUrlHandlerMapping 自然就没发处理静态资源了 但是有RequestMappingHandlerMapping所以可以处理动态资源

下面着重说一下加了""标签后为我们特意提供的RequestMappingHandlerAdapter这个解析器用来替代AnnotationMethodHandlerAdapter 这个过时的解析器 这里我将其作为源码分析 祥见《springMvc源码小析》的RequestMappingHandlerAdapter详解。

数据校验:

数据校验； 只做前端校验是不安全的； 在重要数据一定要加上后端验证； 1.可以写程序将我们每一个数据取出进行校验，如果失败直接来到添加页面，提示其重新填写；x 2.SpringMVC；可以JSR303来做数据校验 JSR303：规范-----Hibernate Validator（第三方校验框架）

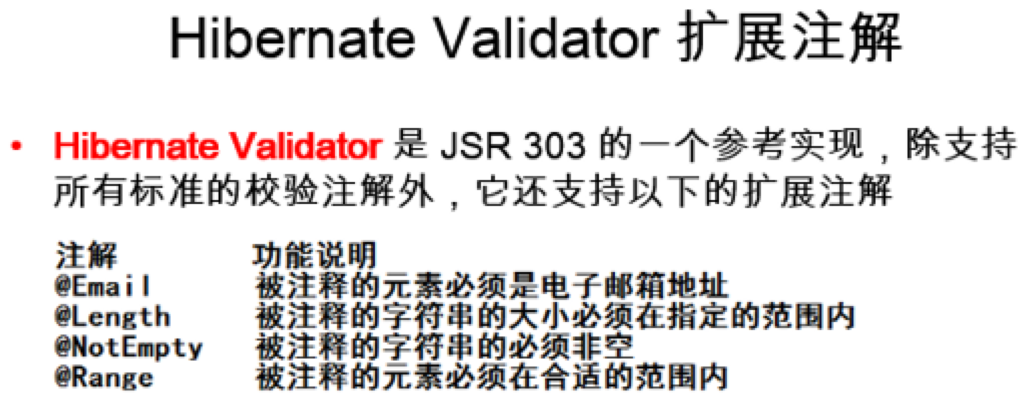


JSR 303

- JSR 303 是 Java 为 Bean 数据合法性校验提供的标准框架，它已经包含在 JavaEE 6.0 中 .
- JSR 303 通过在 Bean 属性上标注类似于 @NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证

注解	功能说明
@Null	被注释的元素必须为 null
@NotNull	被注释的元素必须不为 null
@AssertTrue	被注释的元素必须为 true
@AssertFalse	被注释的元素必须为 false
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期
@Pattern(value)	被注释的元素必须符合指定的正则表达式

注:JSR 303规定的一些校验注解



Hibernate Validator 扩展注解

- Hibernate Validator 是 JSR 303 的一个参考实现，除支持所有标准的校验注解外，它还支持以下的扩展注解

注解	功能说明
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串的必须非空
@Range	被注释的元素必须在合适的范围内

注:Hibernate Validator的一些扩展注解

3.如何快速的进行后端校验； 1.导入校验框架的jar包； 有几个带el的jar不导入；tomcat中有；如果tomcat的版本是 7.0以下 tomcat7.0以上el表达式比较强大； 如果是7.0以下将带el的几个jar放在tomcat的lib文件夹下进行覆盖(建议7.0以上)

```
hibernate-validator-5.0.0.CR2.jar
hibernate-validator-annotation-processor-5.0.0.CR2.jar
classmate-0.8.0.jar
jboss-logging-3.1.1.GA.jar
validation-api-1.1.0.CR1.jar
```

2.只需要给javaBean的属性添加上校验注解

例如:

```java
public class Employee {

    private Integer id;

    @NotEmpty(message="不能为空")
    @Length(min=5,max=17,message="xxx message")
    private String lastName;


    @Email
    private String email;
    //1 male, 0 female
    private Integer gender;


    //规定页面提交的日期格式
    //@Past：必须是一个过去的时间
    //@Future ：必须是一个未来的时间
    //页面提交的数据格式如果不正确，就是400;
    @DateTimeFormat(pattern="yyyy-MM-dd")
    @Past
    @JsonFormat(pattern="yyyy-MM-dd")
    private Date birth = new Date();

    //假设页面，为了显示方便提交的工资是  ￥10,000.98
    @NumberFormat(pattern="#,###.##")
    private Double salary;

    @JsonIgnore
    private Department department;

    //TODO ...
    //省略set get方法
}
```

3.在SpringMVC封装对象的时候，告诉SpringMVC这个javaBean需要校验

例如:

```java
public String addEmp(@Valid Employee employee) {
    //TODO...
}
```

4.如何知道校验结果?

给需要校验的javaBean后面紧跟一个BindingResult。这个BindingResult就是封装前一个bean的校验结果

例如:

```java
//注意:被校验的bean与bindingResult之间一定要连续，不能隔其他的参数
public String addEmp(@Valid Employee employee,BindingResult bindingResult) {
    //TODO...
}
```

5.根据不同的校验结果决定怎么办?

可以根据bindingResult结果进行判断

例如:

```
public String addEmp(@Valid Employee employee,BindingResult bindingResult) {
    boolean errors = bindingResult.hasErrors();
    if(errors) {
        //有错误时处理方法
        //TODO...
    }else {
        //没错误时处理方法
        //TODO...
    }
}
```

如果发生校验错误 ,前台使用的是原生表单，错误信息改如何传达?

将错误放在请求域中就行了

如果将错误放在请求域中，那么需要将那些错误信息放入请求域?

示例如下:

```
public String addEmp(@Valid Employee employee, BindingResult result,
            Model model) {

    // 获取是否有校验错误
    boolean hasErrors = result.hasErrors();
    Map<String, Object> errorsMap = new HashMap<String, Object>();
    if (hasErrors) {
        List<FieldError> errors = result.getFieldErrors();
        for (FieldError fieldError : errors) {
            System.out.println("错误的字段是:" + fieldError.getField());
            System.out.println("错误消息提示: " + fieldError.getDefaultMessage());
            System.out.println(fieldError);
            System.out.println("------------------------");
            //主要是以错误字段为key, 字段错误信息为value 放入map中
            errorsMap.put(fieldError.getField(),
                          fieldError.getDefaultMessage());
        }
        //错误信息放入模型中
        model.addAttribute("errorInfo", errorsMap);

        return "add";
    } else {
        employeeDao.save(employee);
        // 返回列表页面; 重定向到查询所有员工的请求
        return "redirect:/emps";
    }
}
```

如果不想使用springMvc的错误信息，怎么自定义国际化错误信息?

定制自己的国际化错误消息显示,编写国际化的文件;

errors_zh_CN.properties

errors_en_US.properties

每一个字段发生错误以后，都会有自己的错误代码；国际化文件中错误消息的key必须对应一个错误代码,不能乱写

```
codes
[
    Email.employee.email,      校验规则.隐含模型中这个对象的key.对象的属性
    Email.email,                       校验规则.属性名
    Email.java.lang.String,      校验规则.属性类型
    Email
];
从上往下越来越不精确
1.如果是隐含模型中employee对象的email属性字段发生了@Email校验错误，就会生成 Email.employee.email;
2.Email.email: 所有的email属性只要发生了@Email错误; =
3.Email.java.lang.String,:只要是String类型发生了@Email错误
4.Email: 只要发生了@Email校验错误;
```

例如:errors_en_US.properties

```
#同时拥有多个时能使用精确的就用精确的
#Email.email=email incorrect!~~
Email=email buzhengque~~~
NotEmpty=must not empty~~
#可以动态加入一些参数 {0}: 永远都是当前属性名,后面的{1}、{2}才往下排
#例如:@Length(min=5,max=17,message="xxx message") 排序是属性的自然顺序 {1}为max {2}为message {3}为min
Length.java.lang.String= length incorrect ,must between {2} and {1} ~~
Past=must a past time~~~
typeMismatch.birth=birth geshi buzhengque
```

在spring-mvc.xml中配置让springmvc管理国际化资源

```
<!-- 管理国际化资源文件 -->
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
<property name="basename" value="errors"></property>
</bean>
```

再提供一种获取错误信息的方法

```
public class I18TestController {

    @Autowired
    private MessageSource messageSource; ///这个messageSource就是我们上面配置的
ResourceBundleMessageSource类的实例

    /**
    * 这个原生的Locale就是我们下面源码分析时springMvc解析出的那个Locale, spring给你注入进来
    */
    @RequestMapping("/tologinpage")
    public String tologinPage(Locale locale, Model model) {
        System.out.println(locale);
        //可以直接从messageSource中获取到国际化文件中的"xxx"为key的信息
        String xxxinfo = messageSource.getMessage("xxx", null,
                locale);
        System.out.println(xxxinfo);
```

```
        //拿到后放入隐含模型中 这样页面就可以去自行取值了
        model.addAttribute("xxxinfo", xxxinfo);
        return "login";
    }

}
```

说一下国际化的原理:

整个国际化最重要的核心就是区域化信息了 那么springMvc怎么拿到区域化信息的呢?

其实是按照浏览器带来语言信息决定;

Locale locale = request.getLocale();//获取到浏览器的区域信息

在源码中的九大组件有个组件叫LocaleResolver区域化解析器

默认的区域化信息解析器是AcceptHeaderLocaleResolver

所有用到区域信息的地方，都是用到AcceptHeaderLocaleResolver获取的

所以你的浏览器改变语言 区域化信息也会随着改变(原因见下面的源码分析)

```
//区域化信息的源码分析
//区域化解析器的初始化
private void initLocaleResolver(ApplicationContext context) {
    //public static final String LOCALE_RESOLVER_BEAN_NAME = "localeResolver"
    //实际上是我们一般是没有配置localeResolver这个bean的，那么它就会从配置文件中拿
    try {
        this.localeResolver = context.getBean(LOCALE_RESOLVER_BEAN_NAME, LocaleResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using LocaleResolver [" + this.localeResolver + "]");
        }
    }
    catch (NoSuchBeanDefinitionException ex) {
        // We need to use the default.
        //从DispatcherServlet.properties文件中拿
        //org.springframework.web.servlet.LocaleResolver=
        //org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolve这个类
        this.localeResolver = getDefaultStrategy(context, LocaleResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate LocaleResolver with name '" + LOCALE_RESOLVER_BEAN_NAME
+
                    "': using default [" + this.localeResolver + "]");
        }
    }
}
//LocaleResolver区域化解析器接口
public interface LocaleResolver {

    //解析区域化信息
    Locale resolveLocale(HttpServletRequest request);

    //设置区域化信息
    void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale);

}
```

```java
public class AcceptHeaderLocaleResolver implements LocaleResolver {

    private final List<Locale> supportedLocales = new ArrayList<Locale>(4);

    private Locale defaultLocale;


    /**
     * Configure supported locales to check against the requested locales
     * determined via {@link HttpServletRequest#getLocales()}. If this is not
     * configured then {@link HttpServletRequest#getLocale()} is used instead.
     * @param locales the supported locales
     * @since 4.3
     */
    public void setSupportedLocales(List<Locale> locales) {
        this.supportedLocales.clear();
        if (locales != null) {
            this.supportedLocales.addAll(locales);
        }
    }

    /**
     * Return the configured list of supported locales.
     * @since 4.3
     */
    public List<Locale> getSupportedLocales() {
        return this.supportedLocales;
    }

    /**
     * Configure a fixed default locale to fall back on if the request does not
     * have an "Accept-Language" header.
     * <p>By default this is not set in which case when there is "Accept-Language"
     * header, the default locale for the server is used as defined in
     * {@link HttpServletRequest#getLocale()}.
     * @param defaultLocale the default locale to use
     * @since 4.3
     */
    public void setDefaultLocale(Locale defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    /**
     * The configured default locale, if any.
     * @since 4.3
     */
    public Locale getDefaultLocale() {
        return this.defaultLocale;
    }


    @Override
    public Locale resolveLocale(HttpServletRequest request) {
        //拿到默认的区域化信息
        Locale defaultLocale = getDefaultLocale();
        //能拿到并且有"Accept-Language"这个请求头的信息就使用默认的区域化信息
        if (defaultLocale != null && request.getHeader("Accept-Language") == null) {
            return defaultLocale;
        }
        //从request中拿到区域化信息
        Locale requestLocale = request.getLocale();
        //得到所有的可以支持的区域化信息
        List<Locale> supportedLocales = getSupportedLocales();
        //如果此种区域化信息可以支持  就使用request中带来的
        if (supportedLocales.isEmpty() || supportedLocales.contains(requestLocale)) {
            return requestLocale;
        }
        //寻找可以支持的区域化信息
        Locale supportedLocale = findSupportedLocale(request, supportedLocales);
        if (supportedLocale != null) {
            return supportedLocale;
        }
        return (defaultLocale != null ? defaultLocale : requestLocale);
    }
    //查找所有=可以支持的区域化信息
    private Locale findSupportedLocale(HttpServletRequest request, List<Locale> supportedLocales)
{
        //从request中拿出所有的区域化信息
        Enumeration<Locale> requestLocales = request.getLocales();
        Locale languageMatch = null;
        while (requestLocales.hasMoreElements()) {
            Locale locale = requestLocales.nextElement();
            //先根据可支持的区域化信息进行匹配   匹配到了就ok
            if (supportedLocales.contains(locale)) {
                if (languageMatch == null ||
languageMatch.getLanguage().equals(locale.getLanguage())) {
                    // Full match: language + country, possibly narrowed from earlier language-
only match
                    return locale;
                }
            }
            //再按语言的方式去匹配  找到就ok
            else if (languageMatch == null) {
                // Let's try to find a language-only match as a fallback
                for (Locale candidate : supportedLocales) {
                    if (!StringUtils.hasLength(candidate.getCountry()) &&
                            candidate.getLanguage().equals(locale.getLanguage())) {
                        languageMatch = candidate;
                        break;
                    }
                }
            }
        }
        //都没找到返回null
        return languageMatch;
    }
    //设置区域信息   这个实现类不允许修改
    @Override
    public void setLocale(HttpServletRequest request, HttpServletResponse response, Locale locale)
{
        throw new UnsupportedOperationException(
                "Cannot change HTTP accept header - use a different locale resolution strategy");
    }
```

```
}
```

如果想通过自定义按钮进行国际化转换,例如点击中文显示中文页面，点击英文显示英文页面，改怎么办？下面提供两种方法

方法一:自定义区域信息解析器

```
//自定义区域信息解析器 直接实现LocaleResolver接口即可
public class MyLocaleResolver implements LocaleResolver {

    /**
     * 解析返回locale
     */
    @Override
    public Locale resolveLocale(HttpServletRequest req) {
        // TODO Auto-generated method stub
        // zh_CN
        Locale l = null;
        //locale是页面传过来的一个参数  为zh_CN或者en_US
        String localeStr = req.getParameter("locale");
        //如果带了locale参数就用参数指定的区域信息
        if (localeStr != null && !"".equals(localeStr)) {
            l = new Locale(localeStr.split("_")[0], localeStr.split("_")[1]);
        } else {
            //如果没带就用请求头的
            l = req.getLocale();
        }
        return l;
    }

    /**
     * 修改locale  此处表示不支持你修改
     */
    @Override
    public void setLocale(HttpServletRequest arg0, HttpServletResponse arg1,
            Locale arg2) {
        throw new UnsupportedOperationException(
                "Cannot change HTTP accept header - use a different locale resolution strategy");
    }

}
```

配置自定义区域解析器 spring-mvc.xml中配置

```
<!--
ID必须为localeResolver 因为上文我们分析了区域化解析器初始化的过程 可以看到它就是在容器中寻找一个localeResolver的bean 所以不能乱起名
-->
<bean id="localeResolver" class="com.atguigu.controller.MyLocaleResolver"></bean>
```

方法二:将区域信息放入session中

```
//原理 SessionLocaleResolver也是LocaleResolver的一个实现类
public class SessionLocaleResolver extends AbstractLocaleContextResolver {
    public static final String LOCALE_SESSION_ATTRIBUTE_NAME =
SessionLocaleResolver.class.getName() + ".LOCALE";

    private String localeAttributeName = LOCALE_SESSION_ATTRIBUTE_NAME;

    @Override
    public Locale resolveLocale(HttpServletRequest request) {
        //从session中取一个key为SessionLocaleResolver.class.getName() + ".LOCALE"的值 并认为这个就是区域
信息
        Locale locale = (Locale) WebUtils.getSessionAttribute(request, this.localeAttributeName);
        if (locale == null) {
            locale = determineDefaultLocale(request);
        }
        return locale;
    }
}
```

```
@RequestMapping("/tologinpage")
public String tologinPage(@RequestParam(value="locale",defaultValue="zh_CN")String localeStr,
Locale locale,HttpSession session) {


        //如果页面没带locale信息   就给默认的zh_CN
        Locale l = null;
        // 如果带了locale参数就用参数指定的区域信息
        if (localeStr != null && !"".equals(localeStr)) {
            l = new Locale(localeStr.split("_")[0], localeStr.split("_")[1]);
        } else {
            //如果没有就用默认的
            l = locale;
        }
        session.setAttribute(SessionLocaleResolver.class.getName() + ".LOCALE", l);
        return "login";
    }
```

```
<!-- 配置让springMvc使用SessionLocaleResolver这个区域解析器  -->
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
</bean>
```

其实还有方法三使用cookie里放的区域化信息也是可以的其原理来源于CookieLocaleResolver这个解析器类 可以自行查看源码

第四种方式:使用拦截器实现,springMvc实现了一个LocaleChangeInterceptor这个拦截器

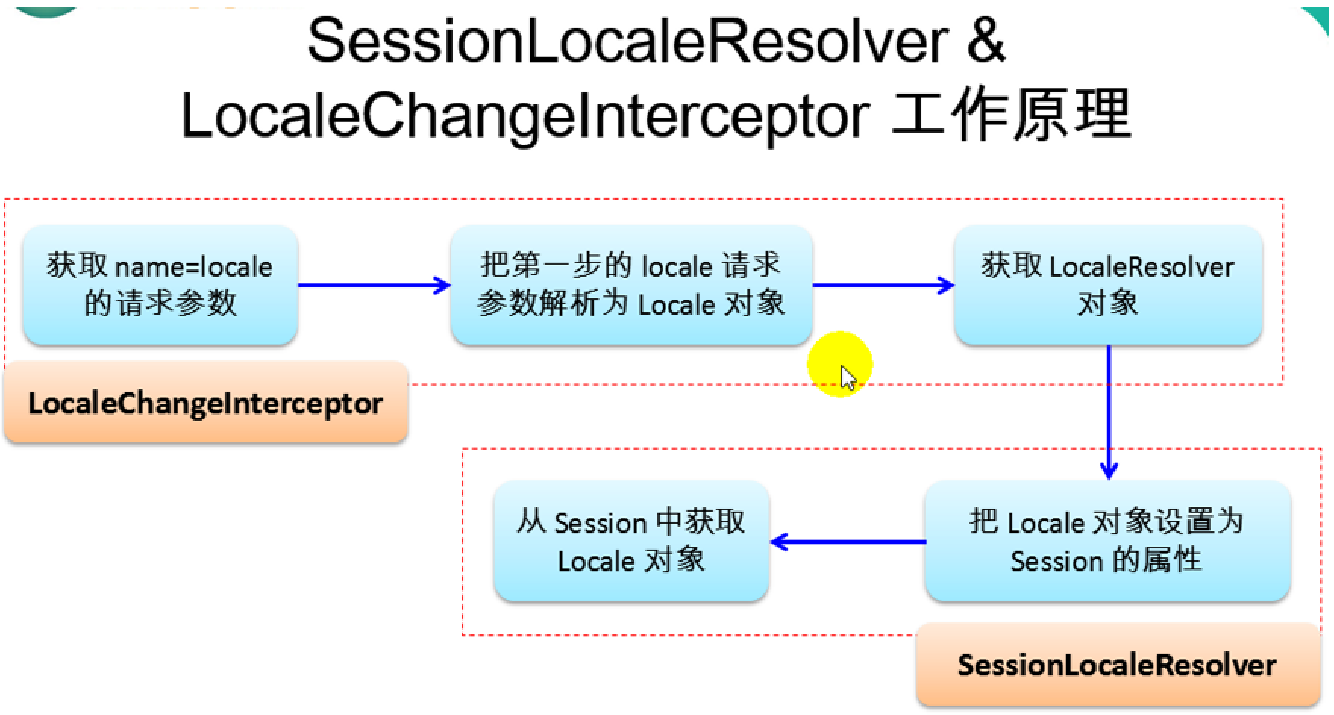
```
//LocaleChangeInterceptor分析

public class LocaleChangeInterceptor extends HandlerInterceptorAdapter {

    /**
     * Default name of the locale specification parameter: "locale".
     */
    public static final String DEFAULT_PARAM_NAME = "locale";
    protected final Log logger = LogFactory.getLog(getClass());

    private String paramName = DEFAULT_PARAM_NAME;

    private String[] httpMethods;

    private boolean ignoreInvalidLocale = false;

    private boolean languageTagCompliant = false;


    /**
     * Set the name of the parameter that contains a locale specification
     * in a locale change request. Default is "locale".
     */
    public void setParamName(String paramName) {
        this.paramName = paramName;
    }

    /**
     * Return the name of the parameter that contains a locale specification
     * in a locale change request.
     */
    public String getParamName() {
        return this.paramName;
    }

    /**
     * Configure the HTTP method(s) over which the locale can be changed.
     * @param httpMethods the methods
     * @since 4.2
     */
    public void setHttpMethods(String... httpMethods) {
        this.httpMethods = httpMethods;
    }

    /**
     * Return the configured HTTP methods.
     * @since 4.2
     */
    public String[] getHttpMethods() {
        return this.httpMethods;
    }

    /**
     * Set whether to ignore an invalid value for the locale parameter.
     * @since 4.2.2
     */
    public void setIgnoreInvalidLocale(boolean ignoreInvalidLocale) {
        this.ignoreInvalidLocale = ignoreInvalidLocale;
    }

    /**
     * Return whether to ignore an invalid value for the locale parameter.
     * @since 4.2.2
     */
    public boolean isIgnoreInvalidLocale() {
```

```java
        return this.ignoreInvalidLocale;
    }

    /**
     * Specify whether to parse request parameter values as BCP 47 language tags
     * instead of Java's legacy locale specification format.
     * The default is {@code false}.
     * <p>Note: This mode requires JDK 7 or higher. Set this flag to {@code true}
     * for BCP 47 compliance on JDK 7+ only.
     * @since 4.3
     * @see Locale#forLanguageTag(String)
     * @see Locale#toLanguageTag()
     */
    public void setLanguageTagCompliant(boolean languageTagCompliant) {
        this.languageTagCompliant = languageTagCompliant;
    }

    /**
     * Return whether to use BCP 47 language tags instead of Java's legacy
     * locale specification format.
     * @since 4.3
     */
    public boolean isLanguageTagCompliant() {
        return this.languageTagCompliant;
    }


    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler)
            throws ServletException {
        //从request中拿到"locale"参数的值 所以需要页面传入"locale"参数
        String newLocale = request.getParameter(getParamName());
        if (newLocale != null) {
            if (checkHttpMethod(request.getMethod())) {
                //获取当前的区域化解析器
                LocaleResolver localeResolver = RequestContextUtils.getLocaleResolver(request);
                //需要先配置区域化解析器，否则直接抛异常
                if (localeResolver == null) {
                    throw new IllegalStateException(
                            "No LocaleResolver found: not in a DispatcherServlet request?");
                }
                try {
                    //设置新的区域信息   但是你配的区域化解析器一定要可以设置区域信息
                    //例如:默认的AcceptHeaderLocaleResolver这个解析器就不行  因为它不允许你设置区域信息
                    //例如:SessionLocaleResolver这个区域解析器就可以
                    localeResolver.setLocale(request, response, parseLocaleValue(newLocale));
                }
                catch (IllegalArgumentException ex) {
                    if (isIgnoreInvalidLocale()) {
                        logger.debug("Ignoring invalid locale value [" + newLocale + "]: " +
ex.getMessage());
                    }
                    else {
                        throw ex;
                    }
                }
            }
        }
        // Proceed in any case.
        return true;
    }

    private boolean checkHttpMethod(String currentMethod) {
        String[] configuredMethods = getHttpMethods();
        if (ObjectUtils.isEmpty(configuredMethods)) {
            return true;
        }
        for (String configuredMethod : configuredMethods) {
            if (configuredMethod.equalsIgnoreCase(currentMethod)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Parse the given locale value as coming from a request parameter.
     * <p>The default implementation calls {@link StringUtils#parseLocaleString(String)}
     * or JDK 7's {@link Locale#forLanguageTag(String)}, depending on the
     * {@link #setLanguageTagCompliant "languageTagCompliant"} configuration property.
     * @param locale the locale value to parse
     * @return the corresponding {@code Locale} instance
     * @since 4.3
     */
    @UsesJava7
    protected Locale parseLocaleValue(String locale) {
        return (isLanguageTagCompliant() ? Locale.forLanguageTag(locale) :
StringUtils.parseLocaleString(locale));
    }

}
```

spring-Mvc.xml中配置拦截器

```
<!-- 配置拦截区域信息改变的拦截器  前提:你需要配置一个拦截器，且这个拦截器可以设置区域信息 默认的
AcceptHeaderLocaleResolver不行 -->
<!-- 所以此处需要手动改变区域化信息解析器 可以配置成我们方法二的那个解析器 -->
<!-- 页面也必须带一个"locale"的参数-->
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
</bean>
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"></bean>
</mvc:interceptors>
```

流程图展示:


SessionLocaleResolver & LocaleChangeInterceptor 工作原理

获取 name=locale 的请求参数
把第一步的 locale 请求参数解析为 Locale 对象
获取 LocaleResolver 对象
LocaleChangeInterceptor
把 Locale 对象设置为 Session 的属性
从 Session 中获取 Locale 对象
SessionLocaleResolver

7.ajax

1.SpringMVC快速的完成ajax功能？ 1.返回数据是json就ok 2.页面,$.ajax() 3.原生javaWeb怎么玩？ 1.导入GSON 2.返回的数据用GSON转成json 3.写出去

```
使用jackson的包
jackson-annotations-2.1.5.jar
jackson-core-2.1.5.jar
jackson-databind-2.1.5.jar
```

示例:

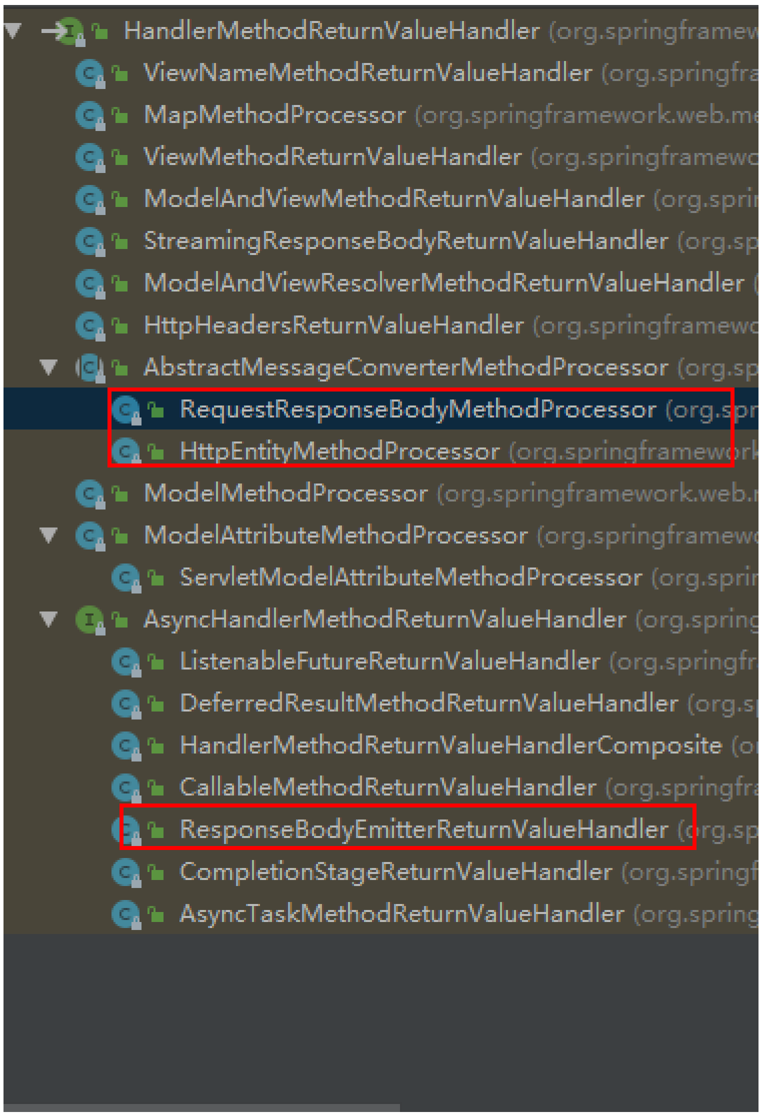
```
public class AjaxTestController {

    @Autowired
    EmployeeDao employeeDao;

    /**
     * SpringMVC文件下载;
     *
     * @param request
     * @return
     * @throws Exception
     */
    @RequestMapping("/download")
    public ResponseEntity<byte[]> download(HttpServletRequest request) throws Exception{

        //1、得到要下载的文件的流;
        //找到要下载的文件的真实路径
        ServletContext context = request.getServletContext();
        String realPath = context.getRealPath("/scripts/jquery-1.9.1.min.js");
        FileInputStream is = new FileInputStream(realPath);

        byte[] tmp = new byte[is.available()];
        is.read(tmp);
        is.close();

        //2、将要下载的文件流返回
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.set("Content-Disposition", "attachment;filename="+"jquery-1.9.1.min.js");

        return new ResponseEntity<byte[]>(tmp, httpHeaders, HttpStatus.OK);
    }

    /**
     * @ResponseBody注解的实质是将返回数据放在响应体中 这也就是既是你返回一个"success"字符串 但是也不会去一个叫
success.jsp的页面
     *
     * ResponseEntity<String>：响应体中内容的类型
     * ResponseEntity更好用，它可以自定义相应头  响应体等
     * @return
     */
    //@ResponseBody
    @RequestMapping("/haha")
    public ResponseEntity<String> hahah(){

        MultiValueMap<String, String> headers = new HttpHeaders();
        String body = "<h1>success</h1>";
        headers.add("Set-Cookie", "username=hahahaha");

        return new ResponseEntity<String>(body , headers, HttpStatus.OK);
    }

    /**
     * 如果参数位置写HttpEntity<String>  str;
     * 比@RequestBody更强，可以拿到请求头；
     *  @RequestHeader("")
     *
     * @param str
     * @return
     */
    @RequestMapping("/test02")
    public String test02(HttpEntity<String>  str){
        System.out.println(str);
        return "success";
    }

    /**
     *
     */
    @RequestMapping("/test01")
    public String test01(@RequestBody String str){
        System.out.println("请求体："+str);
        return "success";
    }

    /**
     *  @RequestBody:请求体；获取一个请求的请求体(用法1)
```

```
      *  @RequestParam:拿到一个请求参数
      *
      *  @ResponseBody: 可以把对象转为json数据, 返回给浏览器(用法2)
      *  @RequestBody: 接受json数据, 可以封装为对象
      * @return
      */
     @RequestMapping("/testRequestBody")
     public String testRequestBody(@RequestBody Employee employee){
         //如果是传的参数是@RequestBody String body
         //则直接拿到@RequestBody中的数据不封装
         System.out.println("请求体: "+employee);
         return "success";
     }

     /**
      * 将返回的数据放在响应体中;
      * 如果是对象, jackson包自动将对象转为json格式
      * @ResponseBody: 可以把对象转为json数据, 返回给浏览器
      * @return
      */
     @ResponseBody
     @RequestMapping("/getallajax")
     public Collection<Employee> ajaxGetAll(){
         Collection<Employee> all = employeeDao.getAll();
         return all;
     }

 }
```

返回值处理器来处理控制器方法执行后的结果 当方法标注了@ResponseBody 或者 ResponseEntity等 其实就是返回值处理器会做相应的处理

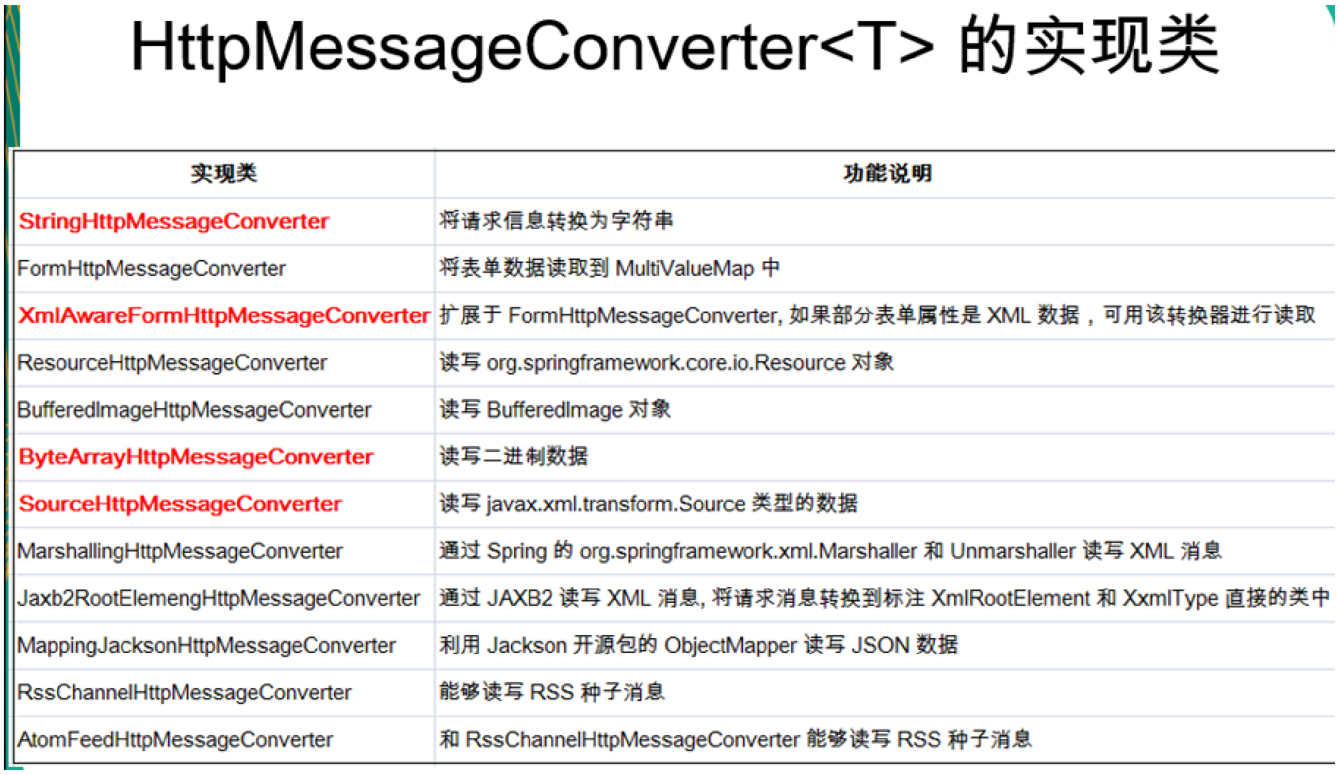
但真正做事的还是一些数据转化器

而无论是@ResponseBody,ResponseEntity 返回值解析器解析，还是@RequestBody, HttpEntity参数解析器解析都是调用了HttpMessageConverter这个接口下的转化器实现的



```
HandlerMethodReturnValueHandler (org.springframew
    ViewNameMethodReturnValueHandler (org.springfra
    MapMethodProcessor (org.springframework.web.me
    ViewMethodReturnValueHandler (org.springframewo
    ModelAndViewMethodReturnValueHandler (org.sprir
    StreamingResponseBodyReturnValueHandler (org.sp
    ModelAndViewResolverMethodReturnValueHandler (
    HttpHeadersReturnValueHandler (org.springframewo
    AbstractMessageConverterMethodProcessor (org.sp
        RequestResponseBodyMethodProcessor (org.spr
        HttpEntityMethodProcessor (org.springframework
    ModelMethodProcessor (org.springframework.web.
    ModelAttributeMethodProcessor (org.springframewo
        ServletModelAttributeMethodProcessor (org.sprin
    AsyncHandlerMethodReturnValueHandler (org.spring
        ListenableFutureReturnValueHandler (org.springfr
        DeferredResultMethodReturnValueHandler (org.s
        HandlerMethodReturnValueHandlerComposite (o
        CallableMethodReturnValueHandler (org.springfra
        ResponseBodyEmitterReturnValueHandler (org.sp
        CompletionStageReturnValueHandler (org.springf
        AsyncTaskMethodReturnValueHandler (org.spring
```

注:HttpMessageConverter主要接口实现类



HttpMessageConverter<T> 的实现类

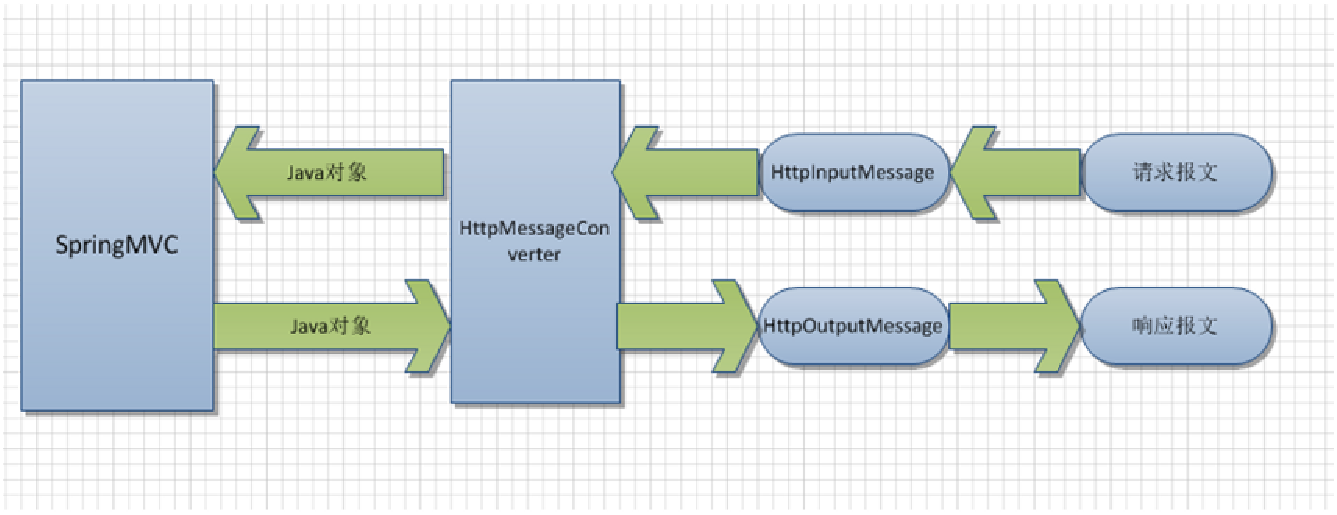
实现类	功能说明
StringHttpMessageConverter	将请求信息转换为字符串
FormHttpMessageConverter	将表单数据读取到 MultiValueMap 中
XmlAwareFormHttpMessageConverter	扩展于 FormHttpMessageConverter, 如果部分表单属性是 XML 数据，可用该转换器进行读取
ResourceHttpMessageConverter	读写 org.springframework.core.io.Resource 对象
BufferedImageHttpMessageConverter	读写 BufferedImage 对象
ByteArrayHttpMessageConverter	读写二进制数据
SourceHttpMessageConverter	读写 javax.xml.transform.Source 类型的数据
MarshallingHttpMessageConverter	通过 Spring 的 org.springframework.xml.Marshaller 和 Unmarshaller 读写 XML 消息
Jaxb2RootElemengHttpMessageConverter	通过 JAXB2 读写 XML 消息, 将请求消息转换到标注 XmlRootElement 和 XxmlType 直接的类中
MappingJacksonHttpMessageConverter	利用 Jackson 开源包的 ObjectMapper 读写 JSON 数据
RssChannelHttpMessageConverter	能够读写 RSS 种子消息
AtomFeedHttpMessageConverter	和 RssChannelHttpMessageConverter 能够读写 RSS 种子消息

注:HttpMessageConverter接口实现类的主要作用

```
//HttpMessageConverter接口类
public interface HttpMessageConverter<T> {

    /**
     * Indicates whether the given class can be read by this converter.
     * @param clazz the class to test for readability
     * @param mediaType the media type to read (can be {@code null} if not specified);
     * typically the value of a {@code Content-Type} header.
     * @return {@code true} if readable; {@code false} otherwise
     */
    //需要被转化的类型可被读?
    boolean canRead(Class<?> clazz, MediaType mediaType);

    /**
     * Indicates whether the given class can be written by this converter.
     * @param clazz the class to test for writability
     * @param mediaType the media type to write (can be {@code null} if not specified);
     * typically the value of an {@code Accept} header.
     * @return {@code true} if writable; {@code false} otherwise
     */
    //需要被转化的类型可被写?
    boolean canWrite(Class<?> clazz, MediaType mediaType);

    /**
     * Return the list of {@link MediaType} objects supported by this converter.
     * @return the list of supported media types
     */
    List<MediaType> getSupportedMediaTypes();
    /**
     * Read an object of the given type from the given input message, and returns it.
     * @param clazz the type of object to return. This type must have previously been passed to
the
     * {@link #canRead canRead} method of this interface, which must have returned {@code true}.
     * @param inputMessage the HTTP input message to read from
     * @return the converted object
     * @throws IOException in case of I/O errors
     * @throws HttpMessageNotReadableException in case of conversion errors
     */
    //需要被转化的类型读转化
    T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
            throws IOException, HttpMessageNotReadableException;

    /**
     * Write an given object to the given output message.
     * @param t the object to write to the output message. The type of this object must have
previously been
     * passed to the {@link #canWrite canWrite} method of this interface, which must have returned
{@code true}.
     * @param contentType the content type to use when writing. May be {@code null} to indicate
that the
     * default content type of the converter must be used. If not {@code null}, this media type
must have
     * previously been passed to the {@link #canWrite canWrite} method of this interface, which
must have
     * returned {@code true}.
     * @param outputMessage the message to write to
     * @throws IOException in case of I/O errors
     * @throws HttpMessageNotWritableException in case of conversion errors
     */
    //需要被转化的类型写转化
    void write(T t, MediaType contentType, HttpOutputMessage outputMessage)
            throws IOException, HttpMessageNotWritableException;

}
```

HttpMessageConverter接口类流程图

HttpMessageConverter<T>



文件上传:

1.文件上传表达准备,主要是enctype="multipart/form-data"以及method="post"

2.导入fileupload的jar包

```
commons-fileupload-1.2.1.jar
commons-io.2.0.jar
```

3.配置

配置spring-mvc.xml

```
<!-- 这个ID不能乱写的 必须为multipartResolver 否则文件上传解析器找不到-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 还有很多参数可设计，详见CommonsFileUploadSupport类 -->
    <!-- 默认下载后存放的位置 -->
    <property name="uploadTempDir" value="D:\\xx\"></property>
    <!-- 默认的最大值 -->
    <property name= "maxUploadSize" value="1024*1024*1024"></property>
    <!-- 默认的编码 -->
    <property name="defaultEncoding" value="utf-8"></property>
</bean>
```

```
//文件上传解析器初始化过程组件  九大组件之一 在DispatcherServlet中
private void initMultipartResolver(ApplicationContext context) {
   try {
       //public static final String MULTIPART_RESOLVER_BEAN_NAME = "multipartResolver";
       //从容器中拿到文件上传的bean给multipartResolver  所以配置时的bean名称不能乱写
      this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME,
MultipartResolver.class);
      if (logger.isDebugEnabled()) {
         logger.debug("Using MultipartResolver [" + this.multipartResolver + "]");
```

```
        }
    }
    catch (NoSuchBeanDefinitionException ex) {
        // Default is no multipart resolver.
         //如果拿不到multipartResolver就为null
        this.multipartResolver = null;
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate MultipartResolver with name '" +
MULTIPART_RESOLVER_BEAN_NAME +
                    "': no multipart request handling provided");
        }
    }
}
```

4.文件上传处理器

```
/**
* @RequestParam(value="username") 接受普通项参数
* MultipartFile file 接受文件项
*/
@RequestMapping("/upload")
public String upload(@RequestParam(value="username") username,@RequestParam(value="img")
MultipartFile file,Model model) throws Exception{

    System.out.println("文件名称:"+file.getName());
    System.out.println("文件原始名称:"+file.getOriginalFileName());


    try {
        //保存文件
        file.transferTo(new File("D:\\xxx\\"+file.getOriginalFileName()));

        model.addAttribute("msg","上传成功");
    }catch(Exception e) {
        model.addAttribute("msg","上传失败");
    }
    return "forward:/success.jsp"
}

/**
* @RequestParam(value="username") 接受普通项参数
* MultipartFile file 接受文件项
* 多文件上传
*/
@RequestMapping("/upload")
public String upload(@RequestParam(value="username") username,@RequestParam(value="img")
MultipartFile[] files,Model model) throws Exception{

    try {
        for (MultipartFile file : files){
            if(!file.isEmpty) {
                System.out.println("文件名称:"+file.getName());
                System.out.println("文件原始名称:"+file.getOriginalFileName());

                //保存文件
                file.transferTo(new File("D:\\xxx\\"+file.getOriginalFileName()));
            }
        }

        model.addAttribute("msg","上传成功");
    }catch(Exception e) {
        model.addAttribute("msg","上传失败");
    }
    return "forward:/success.jsp"
}
```

8.拦截器

SpringMVC提供了拦截器机制；允许运行目标方法之前进行一些拦截工作，或者目标方法运行之后进行一些其他处理；其实和javaweb中的过滤器差不多

Filter；javaWeb定义

HandlerInterceptor：SpringMVC定义

```
//HandlerInterceptorHandlerInterceptor接口
public interface HandlerInterceptor {

    //preHandle：在目标方法运行之前调用；返回boolean；return true；(chain.doFilter())放行；  return
false；不放行
    boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception;

    //postHandle：在目标方法运行之后调用：目标方法调用之后
    void postHandle(
            HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView
modelAndView)
            throws Exception;

    //afterCompletion：在请求整个完成之后；来到目标页面之后；chain.doFilter()放行；资源响应之后
    void afterCompletion(
            HttpServletRequest request, HttpServletResponse response, Object handler, Exception
ex)
            throws Exception;

}
```

1.拦截器是一个接口 2.实现HandlerInterceptor接口；

```
/**
 * 1、实现HandlerInterceptor接口
 * 2、在SpringMVC配置文件中注册这个拦截器的工作；
 *      配置这个拦截器来拦截哪些请求的目标方法；
 *
 *
 */
//第一个拦截器
public class MyFirstInterceptor implements HandlerInterceptor {

    /**
```

```
     * 目标方法运行之前运行
     */
    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception {
        System.out.println("MyFirstInterceptor...preHandle...");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {
        System.out.println("MyFirstInterceptor...postHandle...");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex)
            throws Exception {
        System.out.println("MyFirstInterceptor...afterCompletion");
    }

}
//第二个拦截器
public class MySecondInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("MySecondInterceptor...preHandle...");
        return false;
    }

    @Override
    public void postHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("MySecondInterceptor...postHandle...");

    }

    @Override
    public void afterCompletion(HttpServletRequest request,
            HttpServletResponse response, Object handler, Exception ex)
            throws Exception {
        // TODO Auto-generated method stub
        System.out.println("MySecondInterceptor...afterCompletion...");
    }

}
//处理器
public class InterceptorTestController {

    @RequestMapping("/test01")
    public String test01(){
        System.out.println("test01....");
        //int i =10/0;
        return "success";
    }

}
```

3.配置拦截器

```
<!-- 拦截器  谁先配置谁优先执行preHandle()方法，但是postHandle()和afterCompletion()方法会反过来 -->
<mvc:interceptors>
    <!--配置某个拦截器；默认是拦截所有请求的；  -->
    <bean class="com.mgw.controller.MyFirstInterceptor"></bean>
    <!-- 配置某个拦截器更详细的信息 -->
    <mvc:interceptor>
        <!-- 只来拦截test01请求 -->
        <mvc:mapping path="/test01"/>
        <bean class="com.mgw.controller.MySecondInterceptor"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

4.拦截器的运行流程

正常运行流程；

拦截器的preHandle------目标方法-----拦截器postHandle-----页面-------拦截器的afterCompletion;

```
MyFirstInterceptor...preHandle...
test01....
MyFirstInterceptor...postHandle...
success.jsp....(这是一个页面信息)
MyFirstInterceptor...afterCompletion
```

其他流程：

1.只要preHandle不放行就没有以后的流程；

2.只要放行了，afterCompletion都会执行；

3.目标方法异常，拦截器postHandle不执行，但是拦截器的afterCompletion执行(就像异常中的finally 只要放行了就会执行)

多个拦截器

正常流程：

```
拦截器的配置优先级是谁先配置谁优先执行preHandle()方法，但是postHandle()和afterCompletion()方法会反过来
MyFirstInterceptor...preHandle...
MySecondInterceptor...preHandle...
test01....
MySecondInterceptor...postHandle...(注意这里的顺序 会反过来)
MyFirstInterceptor...postHandle...
success.jsp....(页面信息)
MySecondInterceptor...afterCompletion...(注意这里的顺序 会反过来)
MyFirstInterceptor...afterCompletion
```

异常流程：

1.不放行；

1.哪一块不放行从此以后都没有；

MySecondInterceptor不放行；但是他前面已经放行了的拦截器的afterCompletion总会执行；

(其实就是多拦截器时已放行的拦截器的afterCompletion()总会执行但是postHandle()不一定会执行)

```
MyFirstInterceptor...preHandle...
MySecondInterceptor...preHandle...
MyFirstInterceptor...afterCompletion
```

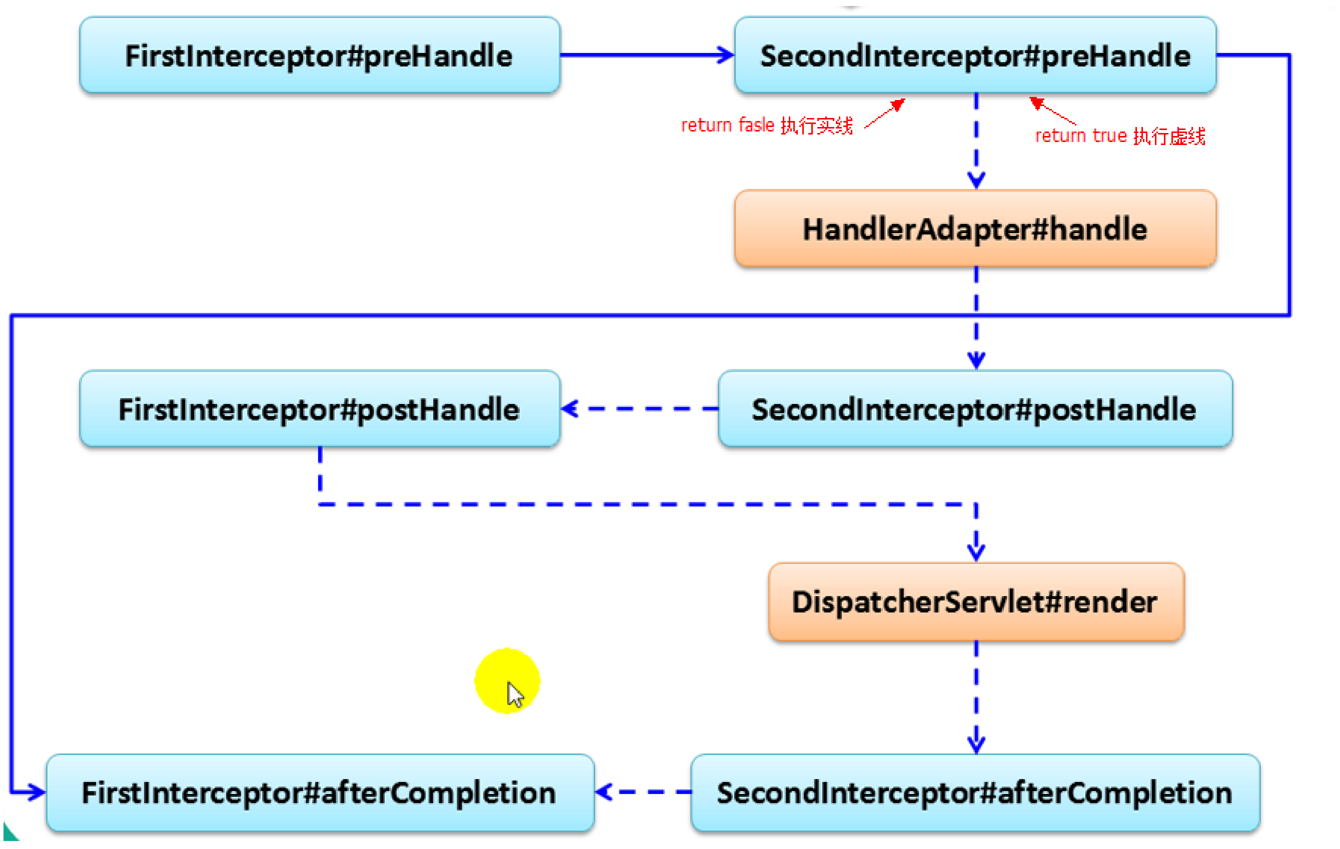
总结:

拦截器的preHandle：是按照顺序执行

拦截器的postHandle：是按照逆序执行

拦截器的afterCompletion：是按照逆序执行；

已经放行了的拦截器的afterCompletion总会执行



FirstInterceptor#preHandle
SecondInterceptor#preHandle
return fasle 执行实线
return true 执行虚线
HandlerAdapter#handle
FirstInterceptor#postHandle
SecondInterceptor#postHandle
DispatcherServlet#render
FirstInterceptor#afterCompletion
SecondInterceptor#afterCompletion

注:拦截器的执行顺序图

拦截器与Filter(过滤器)都可以做方法的拦截:拦截器比Filter更加强大 建议使用拦截器

9.异常处理

springMvc的9大组件之一的HandlerExceptionResolver异常处理解析器 springMvc将异常处理单独作为一个组件， 可见它赋予了这个组件远比原来Exception更加强大的功能。

```
//异常处理解析器初始化
private void initHandlerExceptionResolvers(ApplicationContext context) {
    this.handlerExceptionResolvers = null;

    if (this.detectAllHandlerExceptionResolvers) {
        // Find all HandlerExceptionResolvers in the ApplicationContext, including ancestor
contexts.
        //如果容器中配置了HandlerExceptionResolver接口的实现类 那就拿自己配置的
        Map<String, HandlerExceptionResolver> matchingBeans = BeanFactoryUtils
                .beansOfTypeIncludingAncestors(context, HandlerExceptionResolver.class, true,
false);
        if (!matchingBeans.isEmpty()) {
            this.handlerExceptionResolvers = new ArrayList<HandlerExceptionResolver>
(matchingBeans.values());
            // We keep HandlerExceptionResolvers in sorted order.
            AnnotationAwareOrderComparator.sort(this.handlerExceptionResolvers);
        }
    }
    else {
        try {
            HandlerExceptionResolver her =
                    context.getBean(HANDLER_EXCEPTION_RESOLVER_BEAN_NAME,
HandlerExceptionResolver.class);
            this.handlerExceptionResolvers = Collections.singletonList(her);
        }
        catch (NoSuchBeanDefinitionException ex) {
            // Ignore, no HandlerExceptionResolver is fine too.
        }
    }

    // Ensure we have at least some HandlerExceptionResolvers, by registering
    // default HandlerExceptionResolvers if no other resolvers are found.
    if (this.handlerExceptionResolvers == null) {
        //拿出默认的配置
        /**
        和其他组件初始化拿默认的配置类一样的文件  它拿出的是一下三个类:
        org.springframework.web.servlet.HandlerExceptionResolver=

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\
            org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
            org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver
        */
        this.handlerExceptionResolvers = getDefaultStrategies(context,
HandlerExceptionResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug("No HandlerExceptionResolvers found in servlet '" + getServletName() +
"': using default");
        }
    }
}
```

默认拿出的三个异常解析器类是:

```
AnnotationMethodHandlerExceptionResolver
ResponseStatusExceptionResolver
DefaultHandlerExceptionResolver
```

但是如果你使用了mvc:annotation-driven/mvc:annotation-driven这个配置项

那么会添加一个新的异常解析器ExceptionHandlerExceptionResolver用来替代AnnotationMethodHandlerExceptionResolver

详见上面说过的mvc:annotation-driven>/mvc:annotation-driven配置项究竟做了什么

此时的新的默认的异常解析器是

```
ExceptionHandlerExceptionResolver处理@ExceptionHandler的
ResponseStatusExceptionResolver处理@ResponseStatus
DefaultHandlerExceptionResolver判断是否是springMvc自带的异常
```

异常处理的源码分析:

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // Determine handler for the current request.
            mappedHandler = getHandler(processedRequest);
            if (mappedHandler == null || mappedHandler.getHandler() == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            // Determine handler adapter for the current request.
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

            // Process last-modified header, if supported by the handler.
            String method = request.getMethod();
            boolean isGet = "GET".equals(method);
            if (isGet || "HEAD".equals(method)) {
                long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                if (logger.isDebugEnabled()) {
                    logger.debug("Last-Modified value for [" + getRequestUri(request) + "] is: " +
lastModified);
                }
                if (new ServletWebRequest(request, response).checkNotModified(lastModified) &&
isGet) {
                    return;
                }
            }

            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                return;
            }

            // Actually invoke the handler.
            //如果处理方法发生异常
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            if (asyncManager.isConcurrentHandlingStarted()) {
                return;
            }

            applyDefaultViewName(processedRequest, mv);
            mappedHandler.applyPostHandle(processedRequest, response, mv);
        }
        catch (Exception ex) {
            //此时这里接受到
            dispatchException = ex;
        }
        catch (Throwable err) {
            // As of 4.3, we're processing Errors thrown from handler methods as well,
            // making them available for @ExceptionHandler methods and other scenarios.
            dispatchException = new NestedServletException("Handler dispatch failed", err);
        }
        //进入页面渲染方法中处理
        processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
    }
    catch (Exception ex) {
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
    catch (Throwable err) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
                new NestedServletException("Handler processing failed", err));
    }
    finally {
        if (asyncManager.isConcurrentHandlingStarted()) {
            // Instead of postHandle and afterCompletion
            if (mappedHandler != null) {
                mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
            }
        }
        else {
            // Clean up any resources used by a multipart request.
            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }
}
//processDispatchResult()方法 页面渲染方法
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
            HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) throws
Exception {

    boolean errorView = false;

    //如果有异常先处理异常
    if (exception != null) {
        //如果是模型和视图定义异常则这里处理
        if (exception instanceof ModelAndViewDefiningException) {
            logger.debug("ModelAndViewDefiningException encountered", exception);
            mv = ((ModelAndViewDefiningException) exception).getModelAndView();
        }
        else {
            Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
            //执行处理异常的方法
            mv = processHandlerException(request, response, handler, exception);
            errorView = (mv != null);
        }
    }

    // Did the handler return a view to render?
    if (mv != null && !mv.wasCleared()) {
        render(mv, request, response);
        if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
    }
    else {
        if (logger.isDebugEnabled()) {
            logger.debug("Null ModelAndView returned to DispatcherServlet with name '" +
getServletName() +
                    "': assuming HandlerAdapter completed request handling");
        }
    }

    if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
        // Concurrent handling started during a forward
        return;
    }

    if (mappedHandler != null) {
        mappedHandler.triggerAfterCompletion(request, response, null);
    }
}
//processHandlerException() 执行异常处理的方法
protected ModelAndView processHandlerException(HttpServletRequest request, HttpServletResponse
response,
            Object handler, Exception ex) throws Exception {

    // Check registered HandlerExceptionResolvers...
    ModelAndView exMv = null;
    //循环遍历异常解析器 这个异常解析器就是我们上面说的那个默认的三个 当可以处理时就处理  不能处理时进行找下一个处理
器处理
    for (HandlerExceptionResolver handlerExceptionResolver : this.handlerExceptionResolvers) {
        //将异常信息解析为一个ModelAndView
        exMv = handlerExceptionResolver.resolveException(request, response, handler, ex);
        if (exMv != null) {
            break;
        }
    }
    //如果可以被解析器解析
    if (exMv != null) {
        //且解析信息为空
        if (exMv.isEmpty()) {
            //将异常信息直接放入request域中
            request.setAttribute(EXCEPTION_ATTRIBUTE, ex);
            return null;
        }
        // We might still need view name translation for a plain error model...
        //解析信息不为空
        if (!exMv.hasView()) {
            //设置异常信息视图名称 表明异常信息可以被视图化
            exMv.setViewName(getDefaultViewName(request));
        }
        if (logger.isDebugEnabled()) {
            logger.debug("Handler execution resulted in exception - forwarding to resolved error
view: " + exMv, ex);
        }
        //暴露错误的信息的attribute
        WebUtils.exposeErrorRequestAttributes(request, ex, getServletName());
        return exMv;
    }
    //不能解析再抛 例如数学异常
    throw ex;
}
```

@ExceptionHandler注解的使用

```
public class ExceptionTestController {
    /**
    * 当此处理器发生异常时 可以在此类中写一个异常处理函数
    */
    @RequestMapping("/handle01")
    public String handle01(Integer i) {
        System.out.println("handle01....");
        System.out.println(10 / i);
        return "success";
    }
    //在本类中写一个异常处理函数

    /**
     * 告诉SpringMVC这个方法专门处理这个类发生的异常
     * 1、给方法上写一个Exception类，一种类接受一种异常，类的范围越小接受的异常就越精确,如果直接使用Exception类,
那么当异常发生时如果没有          精确匹配到，就会使用这个异常处理器处理
     * 2、要携带异常信息不能给参数位置写Model;  3、返回ModelAndView就行了;
     * 4、如果有多个@ExceptionHandler都能处理这个异常，精确优先 5、全局异常处理与本类同时存在，本类优先;
     */
    @ExceptionHandler(value = { Exception.class })
    public ModelAndView handleException01(Exception exception) {
        System.out.println("本类的: handleException01..." + exception);
        //
        ModelAndView view = new ModelAndView("myerror");
        view.addObject("ex", exception);
        // 视图解析器拼串
        return view;
    }
}

/**
 * 集中处理所有异常
 * @author lfy
 *
 * 1、集中处理所有异常的类加入到ioc容器中
 * 2、@ControllerAdvice专门处理异常的类
 */
@ControllerAdvice
public class MyAllException {

    @ExceptionHandler(value={ArithmeticException.class})
    public ModelAndView handleException01(Exception exception){
        System.out.println("全局的: handleException01..."+exception);
        //
        ModelAndView view = new ModelAndView("myerror");
        view.addObject("ex", exception);
        //视图解析器拼串
        return view;
    }

    @ExceptionHandler(value={Exception.class})
    public ModelAndView handleException02(Exception exception){
        System.out.println("全局的: handleException02..."+exception);
        //
        ModelAndView view = new ModelAndView("myerror");
        view.addObject("ex", exception);
        //视图解析器拼串
        return view;
    }

}
```

@ResponseStatus注解的使用(这个是给自定义异常标注的) 这个注解不能标注在方法上 否则就算方法正常运行 也一定会返回错误信息//

```
//模拟登录时异常
public class ExceptionTestController {
    @RequestMapping("/handle02")
    public String handle02(@RequestParam("username") String username) {
        if (!"admin".equals(username)) {

            System.out.println("登陆失败....");
            throw new UserNameNotFoundException();
        }
        System.out.println("登陆成功! 。。。");
        return "success";
    }
}
//自定义异常类
@ResponseStatus(reason="用户被拒绝登陆",value=HttpStatus.NOT_ACCEPTABLE)
```

```
public class UserNameNotFoundException extends RuntimeException {

    private static final long serialVersionUID = 1L;

}
```

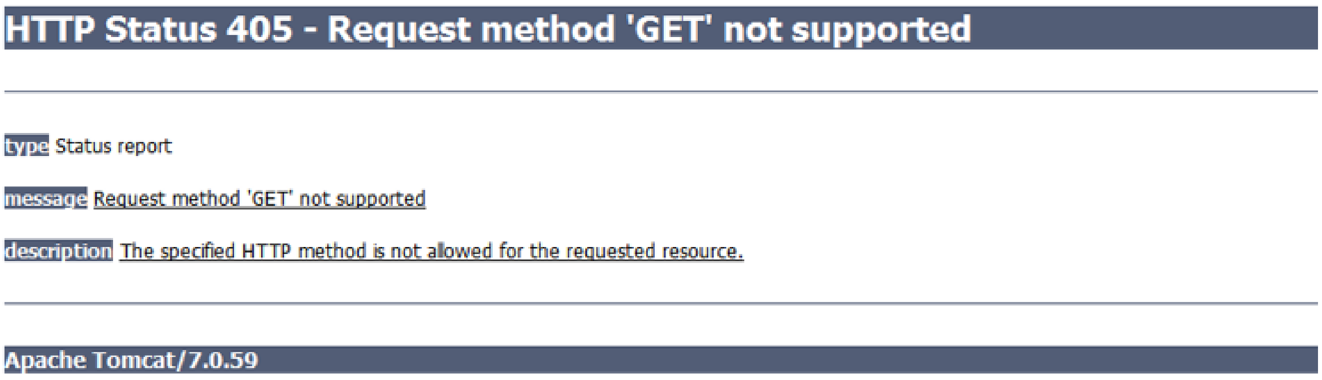
DefaultHandlerExceptionResolver处理springMvc自身的异常

```
public class ExceptionTestController {
    //当发送使用GET提交时就会引发异常
    @RequestMapping(value="/handle03",method=RequestMethod.POST)
    public String handle03(){
        return "success";
    }
}
```

DefaultHandlerExceptionResolver:判断是否SpringMVC自带的异常

Spring自己的异常：如：HttpRequestMethodNotSupportedException。如果没人处理

此时发生异常,DefaultHandlerExceptionResolver处理(特别注意：异常处理器是循环进行处理的，当前面的异常处理器不能处理的时候才轮到DefaultHandlerExceptionResolver处理)


HTTP Status 405 - Request method 'GET' not supported

type Status report

message Request method 'GET' not supported

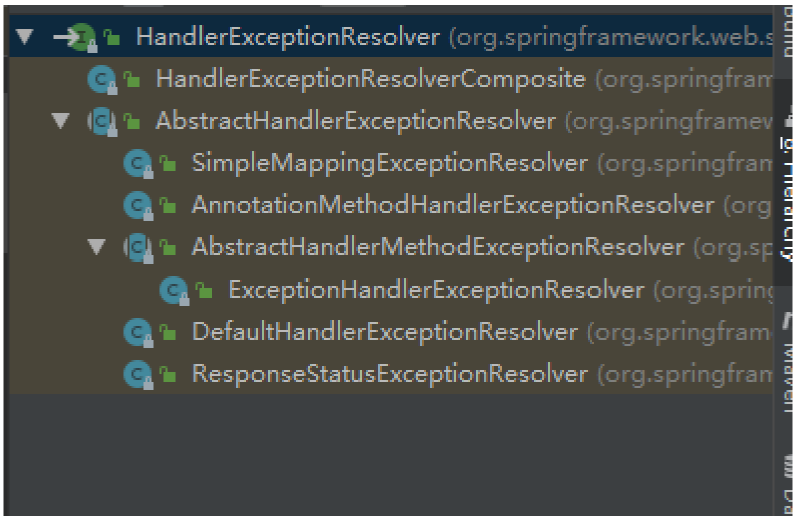
description The specified HTTP method is not allowed for the requested resource.

Apache Tomcat/7.0.59

```
//DefaultHandlerExceptionResolver源码部分展示 表明springMvc可处理的异常
public class DefaultHandlerExceptionResolver extends AbstractHandlerExceptionResolver {
    @Override
    @SuppressWarnings("deprecation")
    protected ModelAndView doResolveException(
            HttpServletRequest request, HttpServletResponse response, Object handler, Exception
ex) {

        try {
            if (ex instanceof
org.springframework.web.servlet.mvc.multiaction.NoSuchRequestHandlingMethodException) {
                return handleNoSuchRequestHandlingMethod(

(org.springframework.web.servlet.mvc.multiaction.NoSuchRequestHandlingMethodException) ex,
                        request, response, handler);
            }
            else if (ex instanceof HttpRequestMethodNotSupportedException) {
                return handleHttpRequestMethodNotSupported(
                        (HttpRequestMethodNotSupportedException) ex, request, response, handler);
            }
```

```java
            else if (ex instanceof HttpMediaTypeNotSupportedException) {
                return handleHttpMediaTypeNotSupported(
                        (HttpMediaTypeNotSupportedException) ex, request, response, handler);
            }
            else if (ex instanceof HttpMediaTypeNotAcceptableException) {
                return handleHttpMediaTypeNotAcceptable(
                        (HttpMediaTypeNotAcceptableException) ex, request, response, handler);
            }
            else if (ex instanceof MissingPathVariableException) {
                return handleMissingPathVariable(
                        (MissingPathVariableException) ex, request, response, handler);
            }
            else if (ex instanceof MissingServletRequestParameterException) {
                return handleMissingServletRequestParameter(
                        (MissingServletRequestParameterException) ex, request, response, handler);
            }
            else if (ex instanceof ServletRequestBindingException) {
                return handleServletRequestBindingException(
                        (ServletRequestBindingException) ex, request, response, handler);
            }
            else if (ex instanceof ConversionNotSupportedException) {
                return handleConversionNotSupported(
                        (ConversionNotSupportedException) ex, request, response, handler);
            }
            else if (ex instanceof TypeMismatchException) {
                return handleTypeMismatch(
                        (TypeMismatchException) ex, request, response, handler);
            }
            else if (ex instanceof HttpMessageNotReadableException) {
                return handleHttpMessageNotReadable(
                        (HttpMessageNotReadableException) ex, request, response, handler);
            }
            else if (ex instanceof HttpMessageNotWritableException) {
                return handleHttpMessageNotWritable(
                        (HttpMessageNotWritableException) ex, request, response, handler);
            }
            else if (ex instanceof MethodArgumentNotValidException) {
                return handleMethodArgumentNotValidException(
                        (MethodArgumentNotValidException) ex, request, response, handler);
            }
            else if (ex instanceof MissingServletRequestPartException) {
                return handleMissingServletRequestPartException(
                        (MissingServletRequestPartException) ex, request, response, handler);
            }
            else if (ex instanceof BindException) {
                return handleBindException((BindException) ex, request, response, handler);
            }
            else if (ex instanceof NoHandlerFoundException) {
                return handleNoHandlerFoundException(
                        (NoHandlerFoundException) ex, request, response, handler);
            }
            else if (ex instanceof AsyncRequestTimeoutException) {
                return handleAsyncRequestTimeoutException(
                        (AsyncRequestTimeoutException) ex, request, response, handler);
            }
        }
        catch (Exception handlerException) {
            if (logger.isWarnEnabled()) {
                logger.warn("Handling of [" + ex.getClass().getName() + "] resulted in exception",
handlerException);
            }
        }
        return null;
    }
}
```

HandlerExceptionResolver接口的实现类展示



```
HandlerExceptionResolver (org.springframework.web.s
    HandlerExceptionResolverComposite (org.springfram
    AbstractHandlerExceptionResolver (org.springframew
        SimpleMappingExceptionResolver (org.springfram
        AnnotationMethodHandlerExceptionResolver (org
        AbstractHandlerMethodExceptionResolver (org.sp
            ExceptionHandlerExceptionResolver (org.spring
        DefaultHandlerExceptionResolver (org.springfram
        ResponseStatusExceptionResolver (org.springfram
```

SimpleMappingExceptionResolver这种异常解析器是支持注解的形式

可以这么使用，在配置文件中为其配置，当发生相应异常时直接调用处理方式即可

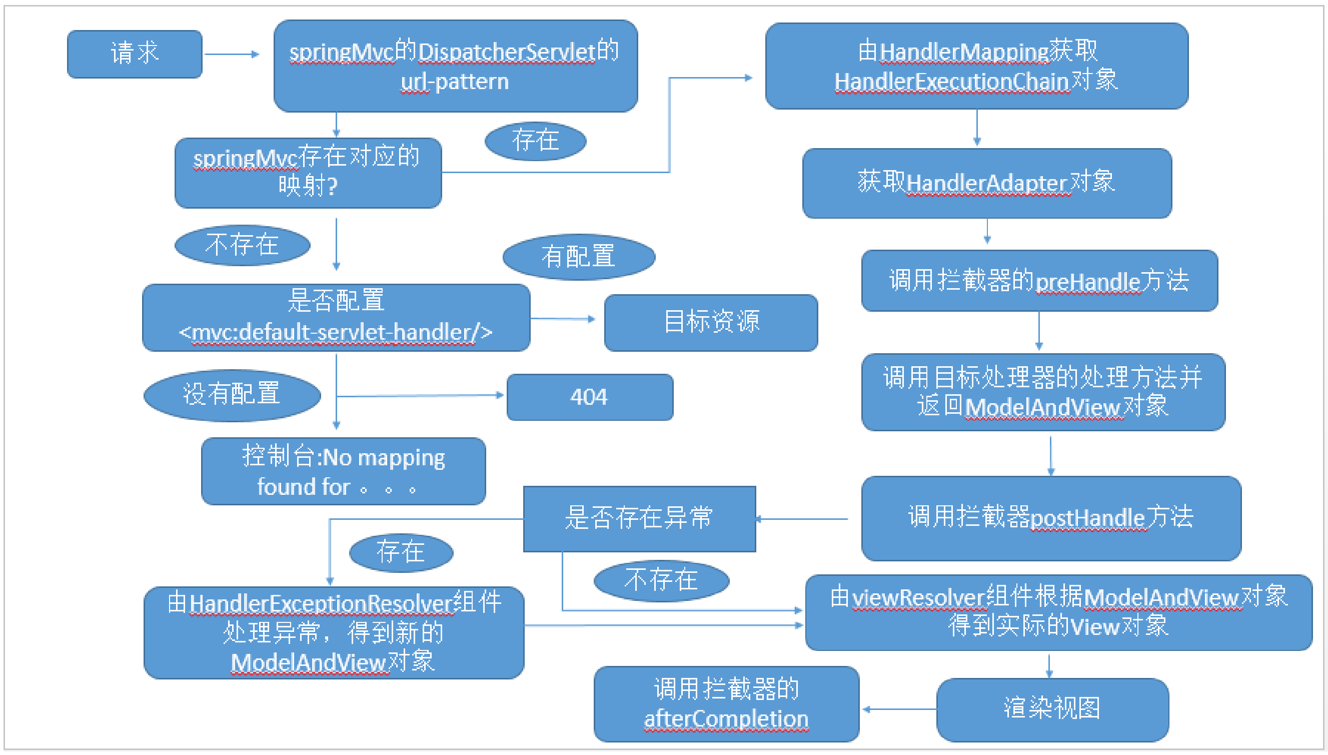
```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <!-- exceptionMappings: 配置哪些异常去哪些页面 -->
    <property name="exceptionMappings">
        <props>
            <!-- key: 异常全类名; value: 要去的页面视图名;  -->
            <prop key="java.lang.NullPointerException">myerrorNullPointer</prop>
            <prop key="java.lang.ArithmeticException">myerrorArithmetic</prop>
        </props>
    </property>
    <!-- 指定错误信息取出时使用的key  -->
    <property name="exceptionAttribute" value="ex"></property>
</bean>
<!-- 注:这个配置的异常处理器优先级极地 若前面默认的异常解析器可以处理，那么我们配置的是不会处理的 -->
<!-- 例如发生了java.lang.ArithmeticException异常 若前面的默认处理器可以处理 那么这里就不会处理 -->
```

```
//测试方法
public class ExceptionTestController {
    @RequestMapping("/handle04")
    public String handle04(){
        System.out.println("handle04");
        String str = null;
        System.out.println(str.charAt(0));
        return "success";
    }
}
```

10.SpringMVC的运行流程

```
SpringMVC运行流程：
1.所有请求，前端控制器（DispatcherServlet）收到请求，调用doDispatch进行处理
2.根据HandlerMapping中保存的请求映射信息找到，处理当前请求的，处理器执行链（包含拦截器）
3.根据当前处理器找到他的HandlerAdapter（适配器）
4.拦截器的preHandle方法先执行
5.适配器执行目标方法，并返回ModelAndView
    1.ModelAttribute注解标注的方法提前运行
    2.执行目标方法的时候（确定目标方法用的参数）
        1.有注解
        2.没注解：
            1.看是否Model、Map以及其他的
            2.如果是自定义类型
                1.从隐含模型中看有没有，如果有就从隐含模型中拿
                2.如果没有，再看是否SessionAttributes标注的属性，如果是从Session中拿，如果拿不到会抛异常
                3.都不是，就利用反射创建对象
6.拦截器的postHandle方法执行
7.处理结果；（页面渲染流程）
    1.如果有异常使用异常解析器处理异常；处理完后还会返回ModelAndView
    2.调用render进行页面渲染
        1.视图解析器根据视图名得到视图对象
        2.视图对象调用render方法
    3.执行拦截器的afterCompletion方法
```

流程图：


请求
springMvc的DispatcherServlet的url-pattern
springMvc存在对应的映射?
存在
由HandlerMapping获取HandlerExecutionChain对象
获取HandlerAdapter对象
不存在
有配置
是否配置<mvc:default-servlet-handler/>
目标资源
调用拦截器的preHandle方法
没有配置
404
调用目标处理器的处理方法并返回ModelAndView对象
控制台:No mapping found for 。。。
是否存在异常
调用拦截器postHandle方法
存在
不存在
由HandlerExceptionResolver组件处理异常，得到新的ModelAndView对象
由viewResolver组件根据ModelAndView对象得到实际的View对象
调用拦截器的afterCompletion
渲染视图

11.SpringMVC与Spring整合

SpringMVC和Spring整合的目的；分工明确；

SpringMVC的配置文件就来配置和网站转发逻辑以及网站功能有关的（视图解析器，文件上传解析器，支持ajax，xxx）；

Spring的配置文件来配置和业务有关的（事务控制，数据源，xxx）；

这里你需要先看一下《三种情况的springMvc的IOC容器的创建过程》这篇里提到的三种方式下容器的创建

在里面同时配置了DispatcherServlet和ContextLoaderListener的情况下会创建一个父子容器

但是如果你同时在配置容器的spring.xml中和spring-mvc.xml中配置了相同的包扫描路径

```
例如:
<context:component-scan base-package="com.mgw"></context:component-scan>
```

那么这个路径下只要是放入容器的所有的bean都会被创建两次,父容器中创建一次,子容器中创建一次

所以我们需要让父子容器分开装配bean，达到各司其职的目的

```
<!-- spring.xml 父容器 -->
<context:component-scan base-package="com.mgw">
        <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
        <context:exclude-filter type="annotation"
expression="org.springframework.web.bind.annotation.ControllerAdvice"/>
</context:component-scan>
```

```
<!-- spring-Mvc.xml 子容器 -->
<context:component-scan base-package="com.atguigu" use-default-filters="false">
        <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
        <context:include-filter type="annotation"
expression="org.springframework.web.bind.annotation.ControllerAdvice"/>
</context:component-scan>
```

注意:父容器是拿不到子容器里的bean的